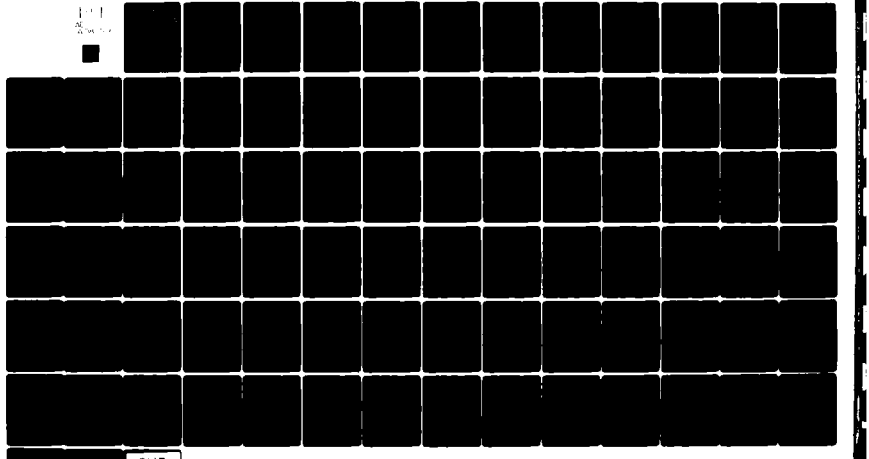
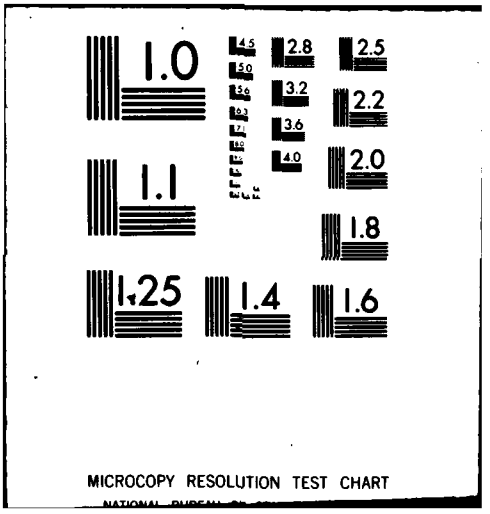
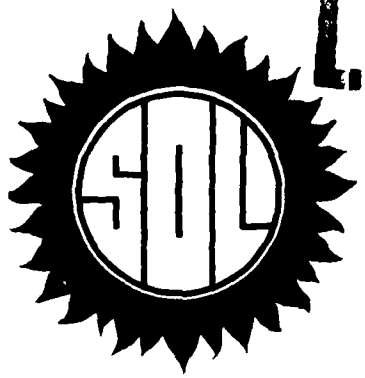
AD-A081 713 STANFORD UNIV CALIF SYSTEMS OPTIMIZATION LAB F/G 12/1
SOLVING STAIRCASE LINEAR PROGRAMS BY THE SIMPLEX METHOD, 1: INV--ETC(U)
NOV 79 R FOURER N00014-75-C-0267
UNCLASSIFIED SOL-79-18 NL



END
DATE FILMED
4-80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART



LEVEL II

Systems
Optimization
Laboratory

Department of Operations Research
Stanford University
Stanford, CA 94305

(12)

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
STANFORD UNIVERSITY
STANFORD, CALIFORNIA
94305

(9) Technical Report

(6) SOLVING STAIRCASE LINEAR PROGRAMS BY THE SIMPLEX METHOD, 1: INVERSION

by

(10) Robert Fourer

(14) TECHNICAL REPORT SOL-79-18
November 1979

(11) Nov 79 (12) 84

(15) N00014-75-C-0267.
DE-AS03-76-SF00326

Research and reproduction of this report were partially supported by the Department of Energy Contract DE-AS03-76-SF00326 PA# DE-AT-03-76ER72018; the Office of Naval Research Contract N00014-75-C-0267; the National Science Foundation Grants MCS76-20019 A01 and ENG77-06761.

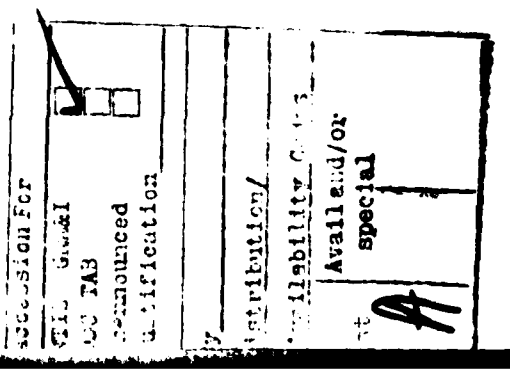
INTRODUCTION

Staircase-structured linear programs (LPs) have been studied about as long as linear programming itself. Staircase LPs arose naturally from models of economic planning over time: activities were run in a series of periods, and constraints linked activities in adjacent periods. The resulting LPs, in their simplest form, had a structure like this:

$$\begin{aligned}
\text{maximize} \quad & c_1x_1 + c_2x_2 + c_3x_3 + \cdots + c_{t-1}x_{t-1} + c_tx_t \\
\textit{subject to} \quad & A_{11}x_1 = b_1 \\
& A_{21}x_1 + A_{22}x_2 = b_2 \\
& A_{32}x_2 + A_{33}x_3 = b_3 \\
& \cdots\cdots\cdots \\
& A_{t,t-1}x_{t-1} + A_{tt}x_t = b_t
\end{aligned}$$

In the infancy of computers this sort of structured problem was attractive because it seemed to offer a hope of solving practical LPs in a reasonable amount of time. Thus in 1949 Dantzig observed [5] that

> ...while the general mathematical problem is concerned with maximization of a linear form of nonnegative variables subject to a system of linear equalities, in the linear programming case one finds by observing the above [staircase] system that the grand matrix of coefficients is composed mostly of blocks of zeros except for submatrices along and just off the "diagonal". Thus any good computational technique for solving programs would probably take advantage of this fact.



The simplex method was as yet impossibly slow for large general problems, but there was reason to think that a much faster version could be devised for staircase LPs.

Staircase linear programs are of no less interest today. Along with economic planning, they have found applications in production scheduling, inventory, transportation, control, and design of multi-stage structures [32]. Yet a recent survey [18] observes that

> the "staircase" model, in which similar sets of variables and constraints are replicated many times, seems no more tractable today then when its importance was recognized over 20 years ago. Typical of many "time-phased" economic problems, it is the standard model for numerically solving problems of optimal control. Today we know only how to solve it as we would any linear programming problem; but this type of problem requires more work to solve than does the average problem of the same size. However, there should be some way to take advantage of their simple structure.

Thus the situation has been reversed. The general simplex method is now impressively fast rather than impossibly slow, while staircase LPs are a troublesomely hard case rather than a promisingly easy one.

Proposed methods for staircase LPs

There has certainly been no shortage of attempts to solve staircase LPs more efficiently. Although the simplex method has usually been involved in some guise, individual proposals have varied considerably. The essential ideas of these proposals may be classified in four broad areas:

Compact basis methods employ a special representation of the basis or basis inverse in conjunction with a more or less standard simplex method. This approach was first suggested by Dantzig [6,8], and early variations were employed by Heesterman and Sandee [23] and Saigal [46]. More recent compact-basis schemes have been worked out by Dantzig [9], Wollmer [51], Marsten and Shepardson [35], Perold and Dantzig [42], and Propoi and Krivonozhko [43].

Nested decomposition methods apply the Dantzig-Wolfe decomposition principle to generate a series of sub-problems at each period. This approach was suggested by Dantzig and Wolfe in their original paper on decomposition [10], and has been extended or modified by Cobb and Cord [4], Glassey [19,20] and Ho and Manne [29]. (Ho has reported favorable computational results in two special cases [26,27].)

Transformation methods start with a simpler LP that can be solved easily, and work toward a solution of the original staircase LP. Varied proposals in this class are from Grinold [22], Aonuma [1], and Marsten and Shepardson [35].

Continuous methods deal with a multi-period LP in continuous rather than discrete time. Fundamentals of a simplex method for continuous-time LPs have been proposed by Perold [41].

Computational experience with most of these proposals is negligible. At present no method has proved as effective as the general simplex method in handling a wide variety of staircase problems.

Adaptation of the simplex method for staircase LPs

Proposals for improving the general simplex method itself have been, by contrast, much more successful. As a result the simplex method has become an amalgam of fairly sophisticated algorithms. Many of these algorithms are objects of study in their own right, and are not normally thought of in connection with linear programming. The simplex method has consequently become more and more a specialist's domain.

It is therefore not surprising that study of staircase LPs has tended to diverge from study of the simplex method. Staircase linear programming, typified by the above-listed papers, has become a search for methods to replace the old simplex method; in the mean time a new, better simplex method has emerged for general linear programming but has not been applied to special structures such as staircases.

This and a companion paper [16] seek to reverse the trend: they are concerned with adapting the modern simplex method to solve staircase LPs more efficiently. Each paper looks at a set of algorithms within the simplex method: this one deals with "inversion" of the basis--more accurately, solution of linear systems by Gaussian elimination--and the succeeding one considers partial pricing.

Both papers describe extensive, although preliminary, computational experience. The results are quite promising: a staircase-adapted simplex method sometimes performs considerably better than the general method, yet on a range of problems it is never significantly worse. Moreover, further improvement appears possible in a number of respects.

1. STAIRCASE LINEAR PROGRAMS

Staircase linear programs share two simple characteristics: their variables fall into some sequence of disjoint groups; and their constraints relate only variables within adjacent groups. Usually the sequence of groups corresponds to a sequence of times, so that variables in a group represent activities during one time period. Constraints then indicate how activities in one period are related to activities in the next. Staircase LPs thus arise especially often from many kinds of economic planning models.

A constraint is said to be in period ℓ if it contains variables of period ℓ but not of later periods. Typically some constraints involve only variables of period ℓ, while others relate variables of periods ℓ and $\ell-1$; the latter are linking constraints, whereas the former are non-linking. Analogously, linking variables appear in constraints of periods ℓ and $\ell+1$, while non-linking variables appear only in constraints of period ℓ.

A staircase LP is also naturally viewed as a kind of linear discrete-time optimal control model. Typically such a model minimizes a linear function of nonnegative state vectors x_ℓ and control vectors u_ℓ, subject to dynamic equations,

$$C_\ell x_{\ell+1} = A_\ell^{(1)} x_\ell + D_\ell^{(1)} u_\ell + b_\ell^{(1)} , \qquad \ell = 1, \ldots , t$$

and control contraints,

$$0 = A_\ell^{(2)} x_\ell + D_\ell^{(2)} u_\ell + b_\ell^{(2)} , \qquad \ell = 1, \ldots , t+1$$

This is readily seen to be a staircase linear program. The state vectors are the linking variables, and the control vectors are the non-linking variables; the dynamic equations are the linking constraints, while the control contraints are non-linking.

Staircase LPs of higher orders

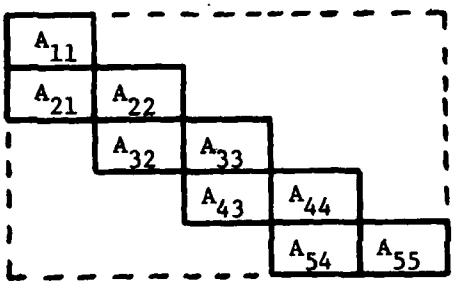
A more general approach says that a staircase linear program is of order r if its constraints relate variables that are at most r periods apart. The preceding subsection thus characterized staircase LPs of order one. Higher-order staircase LPs are not uncommon in complex applications (for example, modeling energy systems [40]). They are analogous to linear control models that have rth-order dynamic equations.

This paper is predominantly concerned with first-order staircase LPs: these have the most specialized structure and, consequently, are most amenable to special techniques. Still, many techniques are essentially applicable to higher-order staircases as well, with appropriate adaptations that will be pointed out as the exposition proceeds. For brevity, however, the adjective "first-order" will usually be dropped.

Higher-order staircase LPs can also be made into first-order ones, in either of two ways. First, rth-order equations can be transformed to equivalent first-order ones by adding certain variables and constraints. This yields a larger first-order LP that has the same number of periods. Second, every r periods of the rth-order LP may simply be aggregated as one period. The result is a first-order staircase LP of the same size but having only about t/r periods. The first method is most practical when the LP is nearly first-order to begin with, while the second may be feasible when the number of periods is large relative to r.

Staircase matrices

The matrix of constraint coefficients of a staircase linear program is a staircase matrix. Its nonzero elements are confined to certain submatrices centered roughly on and just off the diagonal--as, for example,



Formally, one partitions the rows of an $m \times n$ matrix A into t disjoint subsets, and the columns into t disjoint subsets, so that the matrix is partitioned into t^2 submatrices, or "blocks":

$$A_{ij}, \qquad i = 1,\ldots, t; \quad j = 1,\ldots, t$$

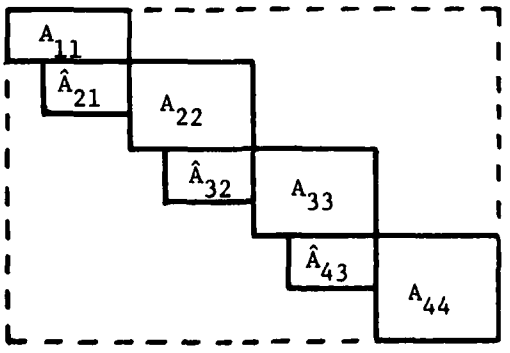
A is lower staircase (as above) if $A_{ij} = 0$ expcet for $i = j$ and $i = j+1$. A is upper staircase if $A_{ij} = 0$ except for $i = j$ and $i = j-1$.

By analogy with staircase models, rows in the ith partition of a staircase matrix A are called period-i rows, and columns in the jth partition are called period-j columns. If a period-i row has nonzero

elements in blocks $A_{i,i-1}$ and A_{ii}, it is a linking row; if it has nonzeroes only in A_{ii} it is a non-linking row. Similarly, a period-j column that has nonzeroes in A_{jj} and $A_{j+1,j}$ is a linking column, while one that has nonzeroes in A_{jj} only is a non-linking column.

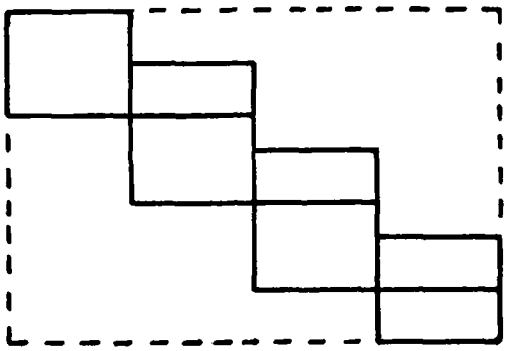
Any upper-staircase matrix may be permuted to lower-staircase form by reversing the order of the periods [15]. Moreover, if a period-i row is entirely zero within A_{ii} that row may be moved back to period i-1 without disrupting the staircase structure; analogously, a period-j column that is all-zero within A_{jj} may be moved to period j+1. Nothing is lost, therefore, in assuming that A is lower staircase and that its diagonal blocks $A_{\ell\ell}$ have no all-zero rows or columns; A is then said to be in <u>standard</u> staircase form. Henceforth it will be assumed that all staircase LPs have a constraint matrix A in this standard form. (The trivial case in which A has an all-zero row or column is thus ruled out.)

Following [15], the period-i rows may be permuted to put the linking rows first, and the period-j columns may be permuted to put the linking columns last. Then A has the following <u>reduced</u> form:



The reduced block $\hat{A}_{k,k-1}$ is just the intersection of the period-k linking rows and the period-(k-1) linking columns.

If the linking rows of every period i are switched to period i-1, then A gains an alternative row-upper-staircase form:



Switching the linking columns of period j to period j+1 gives a different, column-upper-staircase form. Thus a staircase A in reduced standard form embodies three staircases--lower, row-upper, and column-upper--each corresponding to a different choice of where the periods begin and end.

Staircase bases

Any basis B of a staircase linear program necessarily inherits a staircase structure from the constraint matrix A; B's staircase blocks, $B_{\ell,\ell-1}$ and $B_{\ell\ell}$, may be taken to be the sub-blocks of $A_{\ell,\ell-1}$ and $A_{\ell\ell}$ that contain only the basic columns. If A has a reduced form, $\hat{B}_{\ell,\ell-1}$ may likewise be taken as the basic part of $\hat{A}_{\ell,\ell-1}$.

The inherited staircase of B need not be in standard or reduced form, even though A is. Specifically, either $B_{\ell\ell}$ or $\hat{B}_{\ell,\ell-1}$ may be

zero along some linking row i--if it happens that, in $A_{\ell\ell}$ or $\hat{A}_{\ell,\ell-1}$, all the nonzeroes along row i are in non-basic columns. In this event, B may be returned to reduced standard form by reassigning certain rows and columns. Any linking row that is zero in $B_{\ell\ell}$ becomes a non-linking row in period $\ell-1$; in the process, some linking columns of period $\ell-1$ may become non-linking. Any linking row that is zero in $B_{\ell,\ell-1}$ becomes a non-linking row.

It is generally more convenient to deal with B in its inherited staircase form, whether standard, reduced or otherwise. However, better results are often achieved by using B's reduced standard form instead, especially as it has fewer linking rows and columns and hence a tighter structure. This issue is considered further subsequently.

Henceforth $B_{\ell\ell}$ and $B_{\ell,\ell-1}$ (or $\hat{B}_{\ell,\ell-1}$) will represent the blocks of B's chosen staircase form, whether inherited or reduced standard. The number of rows in period i will be denoted m_i, and the number of columns in period j will be n_j; the respective numbers of linking rows and columns will be $\hat{m}_i$ and $\hat{n}_j$. For the row-upper-staircase form, the number of rows in period i will be m^i, and for the column-upper-staircase form the number of columns in period j will be n^j. Necessarily $\sum m_i = \sum m^i = \sum n_j = \sum n^j = m$, and $\hat{m}_i \leq m_i$, $\hat{n}_j \leq n_j$.

Balance constraints and square sub-staircases

If the staircase LP has a special dynamic Leontief structure [7] then in each period the number of basic columns must exactly equal the number of rows: $n_\ell = m_\ell$ for all ℓ, and all blocks $B_{\ell\ell}$ are square.

This is not the case in general, however. A basis B of an arbitrary staircase LP may have $n_\ell > m_\ell$ for some periods ℓ and $n_\ell < m_\ell$ for others.

Since the basis is nonsingular, however, it must obey the "balance constraints" developed in [15]. In summary, these restrict the excess of basic columns over rows in each period, individually and cumulatively, as follows:

$$0 \le \sum_1^\ell (n_i - m_i) \le \min(\hat{m}_{\ell+1}, \hat{n}_\ell), \quad \ell = 1, \ldots, t-1$$

$$-\min(\hat{m}_k, \hat{n}_{k-1}) \le \sum_k^\ell (n_i - m_i) \le \min(\hat{m}_{\ell+1}, \hat{n}_\ell), \quad k, \ell = 2, \ldots, t-1$$

$$-\min(\hat{m}_k, \hat{n}_{k-1}) \le \sum_k^t (n_i - m_i) \le 0 \qquad k = 2, \ldots, t$$

In words, the cumulative imbalance between rows and basic columns in periods k through ℓ is bounded by the smaller dimension of $\hat{B}_{k,k-1}$ and the smaller dimension of $\hat{B}_{\ell+1,\ell}$. Hence these constraints are quite strict when there are relatively few linking rows or columns.

The first constraint above may also be written as the following three inequalities:

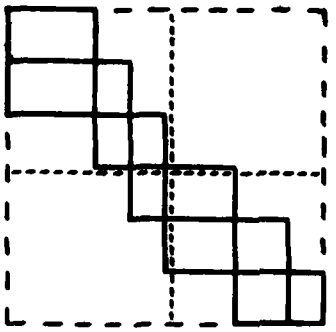
$$\sum_1^\ell n_i \ge \sum_1^\ell m_i$$

$$\sum_1^\ell n_i \le \sum_1^\ell m^i$$

$$\sum_1^\ell n^i \le \sum_1^\ell m_i$$

These say that the first ℓ periods of the lower staircase cannot have more rows than columns, while the first ℓ periods of the associated row-upper or column-upper staircase cannot have more columns than rows.

All three of these relations are equalities when $\ell = t$, since B is square. It can also happen that equality is achieved for some $\ell < t$. For example, if $\sum_1^\ell m_i = \sum_1^\ell n_i$, B must look something like this:



$\sum_1^3 m_i = \sum_1^3 n_i$

The rows and columns of periods 1 through ℓ form a <u>square sub-staircase</u>, as do the rows and columns of periods $\ell+1$ through t; they are linked only by nonzero elements in the off-diagonal block $B_{\ell+1,\ell}$. In a similar way an equality $\sum_1^\ell n_i = \sum_1^\ell m^i$ implies a pair of square sub-staircases within the row-upper staircase form, and $\sum_1^\ell n^i = \sum_1^\ell m_i$ implies the same for the column-upper form.

Generally B may exhibit any or all of these three kinds of equalities, and each may hold for several values of $\ell < t$. If p different such equalities hold, then B breaks into p+1 disjoint square sub-staircases of various kinds. The presence or absence of sub-staircases will be of importance to several of the techniques described further on in this paper.

2. SOLVING LINEAR SYSTEMS IN THE SIMPLEX METHOD

In solving linear programs by the simplex method, a great deal of computational effort is devoted to "inverting the basis". More precisely, at each iteration the simplex method solves two linear systems:

$$By = a$$

$$B^T\pi = z$$

B is the basis, an $m \times m$ matrix of basic columns of the constraint matrix A; a is a non-basic column of A; and z is an appropriately chosen "pricing form".*

There are many ways to solve such systems, but not all are suitable to practical linear programming. Typically m is in the range of several hundred to several thousand, and the simplex method generates roughly $2m$ <u>different</u> bases B. Hence only very efficient solution techniques are useful. Further, B has two very special properties:

- Successive bases are similar. Only one column of B is changed at each iteration.
- Bases are sparse. For a typical large application, less than 1% of the elements of an average B are nonzero.

The best techniques can use these properties to advantage in various ways that are outlined in this section.

---

*It is general practice to incorporate the linear objective function as a row of A. Then, when the basis is feasible, the pricing form z is a unit vector; when the basis is infeasible, z has one nonzero element--either +1 or -1--corresponding to each infeasible basic variable. The exact choice of z depends on details of the implementation, as explained in [39,50].

Permutation of the basis

The variables and equations of a linear system $By = a$ or $B^T\pi = z$ can be written in any order. Each ordering of the variables corresponds to some permutation of the columns of B, while each ordering of the equations corresponds to some permutation of the rows of B.

Any permutation of the rows and columns of B may be written PBQ^T, where P and Q^T are suitably chosen permutation matrices. The system $By = a$ is thus equivalent to the permuted system $(PBQ^T)(Qy) = (Pa)$. $B^T\pi = z$ is likewise equivalent to $(QB^TP^T)(P\pi) = (Qz)$.

LU factorization

At the heart of recent simplex implementations is a technique based on Gaussian elimination. The basis B is factored as the product of a lower-triangular matrix L, and an upper-triangular matrix U. Once $B = LU$ is known, the linear systems of importance reduce to

$$L(Uy) = a$$
$$U^T(L^T\pi) = z$$

Then y or π may be found through solving two triangular systems by back-substitution.

In practice Gaussian elimination is applied to a chosen permutation PBQ^T. Choice of P and Q^T is a crucial matter, as can be seen by considering the computation involved in elimination. Its essential operations are defined by the following recursion:

$$\beta^{(1)} = PBQ^T$$

$$\beta_{ij}^{(k+1)} = \beta_{ij}^{(k)} - \beta_{ik}^{(k)}\beta_{kj}^{(k)}/\beta_{kk}^{(k)}, \qquad i, j > k; \quad k = 1,\ldots, m-1$$

of which L and U are a by-product:

$$L_{ij} = \beta_{ij}^{(j)}/\beta_{jj}^{(j)}, \qquad i \geq j$$

$$U_{ij} = \beta_{ij}^{(i)}, \qquad i \leq j$$

The critical values are the "pivots" $\beta_{kk}^{(k)}$: an LU factorization exists if and only if all pivots are nonzero. Moreover, elimination is numerically stable only if all picots are sufficiently large in magnitude, both absolutely and relative to other elements of $\beta^{(k)}$.

As a consequence, practical Gaussian elimination looks for permutations P and Q^T such that PBQ^T has an acceptably large series of pivots. Choosing P and Q^T is thus commonly called "pivot selection".

Once L and U are computed, solving the resulting triangular systems presents no difficulty. Back-substitution in these systems is an inherently fast and stable process.

The jargon of LP computer codes refers to solution of a lower-triangular system as an FTRAN ("forward transformation"); solution of an upper-triangular system is a BTRAN ("backward transformation"). Solving $L(Uy) = a$ thus requires first an FTRANL and then a BTRANU, while solving $U^T(L^T\pi) = z$ requires an FTRANU and a BTRANL.

Updating the LU factorization

Just as successive bases are similar, their LU factorizations are similar. Consequently it is practical to merely update L and U at each basis change, rather than compute the factorization from scratch each time.

The idea of an LU update is as follows. Suppose the initial basis, B_0, has been factored as $P_0 B_0 Q_0^T = L_0 U_0$. Thus $B_0 = (P_0^T L_0)(U_0 Q_0)$: B_0 is the product of a permuted lower-triangular matrix and a permuted upper-triangular matrix. Equivalently, $(P_0^T L_0)^{-1} B_0 = U_0 Q_0$.

Now update B_0 to a new basis B_1, and consider

$$(P_0^T L_0)^{-1} B_1 = \tilde{U}_0 Q_0 \tag{1}$$

$\tilde{U}_0$ need not be upper-triangular; however, it does have an LU factorization, $\tilde{U}_0 Q_0 = (P_1^T L_1)(U_1 Q_1)$. Substituting into (1) and rearranging shows that

$$B_1 = (P_0^T L_0)(P_1^T L_1)(U_1 Q_1) \tag{2}$$

Thus B_1 is factored as the product of <u>two</u> permuted lower-triangular matrices and a permuted upper-triangular matrix. Linear systems involving B_1 are then readily solved as before, but with the addition of some back-substitutions in L_1.

Similar updates can be applied at subsequent basis changes. After k iterations, the basis B_k is factored at

$$B_k = (P_0^T L_0)(P_1^T L_1) \cdot \cdots \cdot (P_k^T L_k)(U_k Q_k) \tag{3}$$

FTRANL and BRRANL perform back-substitutions with L_0 through L_k, while FTRANU and BTRANU use U_k.

LU updating in this way is practical because B_1 differs from B_0 in only one column. Hence $\tilde{U}_0$ is nearly upper-triangular--it differs from U_0 in only one column--and, as a result, U_1 is much the same as U_0, while L_1 is not much different from the identity. The factorization (2) is thus fairly easy to find and record, and the subsequent back-substitutions are only marginally more expensive than for B_0. Further updates are equally economical, and may continue until the cost of back-substitution in (3) begins to rise appreciably--typically after 50 to 100 iterations. A fresh LU factorization of the basis is then computed, and updating begins anew.

Specific algorithms for LU updates differ primarily in their choice of permutations P_1 and Q_1 for the factorization $U_0Q_0 = (P_1^TL_1)(U_1Q_1)$. The original algorithm of Bartels and Golub [2,3] was designed to ensure numerical stability. Subsequent variations have given more weight to storage arrangement [14,47] or sparsity [17,44].*

*Another technique, proposed by McBride [36], promises an especially sparse update. Essentially, it uses as B_1 a carefully updated and permuted B_0, with the result that the product $(P_0^TL_0)(P_1^TL_1)$ may be collapsed to a single lower-triangular factor; in effect this technique updates the lower-triangular factor at each iteration, whereas the other techniques merely augment it. McBride avoids Gaussian elimination in his implementation, however, preferring to keep the inverse of one small matrix explicitly.

Storing the LU factorization

To benefit from sparsity, an LP code must store only the nonzero elements in matrices such as A, L and U. The total storage required by a sparse problem is thereby drastically curtailed; indeed, large-scale linear programming would be impossible if all zeroes had to be stored. Moreover, sparse storage makes possible efficient pricing and pivoting routines that automatically skip multiplying and adding zeroes.

Because bases are subsets of the columns of A, it is universal practice to store A by column. Typically one array lists the nonzero elements of A in column order, a parallel array lists the row index for each element, and a shorter third array indicates where each column begins in the first two arrays. A basis is represented by just a list of the basic columns.

To factorize a basis B stored in this way, it may be efficient to rearrange the operations of Gaussian elimination so that only one column, b_j, is processed at a time. An LU factorization of PBQ^T is then computed by essentially the following algorithm:

1: SET $L = U = I$

2: REPEAT for each column b_j of BQ^T:

2.1: SOLVE $Lx = Pb_j$ for x

2.2: SET $U_{ij} = x_i$ for $i = 1, \ldots, j$

2.3: SET $L_{ij} = x_i/x_j$ for $i = j+1, \ldots, m$

L and U are produced one column at a time, and so may be stored like A as columnwise lists of nonzeroes. FTRAN operations read forward through

these lists, whereas BTRAN operations start at the end of a list and read backward to the beginning. (Hence the terms FTRAN and BTRAN.)

In practice the storage arrangement of L and U is closely tied to the updating technique. Any of the previously-mentioned techniques may store L columnwise, since it is just augmented (by $P_k^T L_k$) at each iteration. Only the Forrest-Tomlin technique, however, can be adequately implemented with U stored columnwise. Saunders' technique requires row-wise access as well to a (hopefully small) part of U, while Reid's technique is only practical with row-wise access to all of U. Thus these latter techniques have been implemented with various alternative storage schemes for U: Saunders has stored part of U explicitly [47], while Reid has experimented both with linked lists and with a combination of row-wise and column-wise arrays [45].

There are important advantages to storing L and U by column only. Column-wise storage is simple and compact; the associated FTRAN and BTRAN routines are also simple and L and U may be held on any sequential storage device. In a virtual-machine environment, sequential storage also minimizes the danger of "thrashing"--excessive overhead cost that results from trying to access too many widely-separated parts of storage in a short interval of time. On the other hand, if storage is at a premium one may take further advantage of "triangle" columns--those that are zero above the diagonal of PBQ^T; a triangle column is essentially trivial in U and unchanged in L, and so may be represented in L by just a pointer into A.

Access to U by column only does have its disadvantages, however. It restricts updating to the Forrest-Tomlin technique which, while usually

adequate, is inferior to other techniques in numerical stability and sparsity. In addition, it suffers from certain inefficiencies in applying FTRAN and BTRAN to sparse vectors, as explained further below.

Sparse LU factorization

It is well-known [11,12,13] that when B is sparse, some of its permutations have much sparser L and U factors than others. Consequently all LP codes implement some form of sparse Gaussian elimination in which pivots are chosen to promote sparsity of L and U as well as numerical stability.

There are principally two techniques of sparse Gaussian elimination employed in linear programming. Bump-and-spike techniques look for a block-triangular permutation of B that has many small blocks ("bumps") and few columns ("spikes") that extend above the diagonal. Local-minimization techniques choose each pivot to minimize the estimated number of nonzeroes added to L and U by that pivot alone. These ideas are described and compared in Section 1 of [15].

Each technique of sparse elimination is best suited to certain updating techniques. Saunders' update relies on there being relatively few spikes in U, and so it has been implemented with bump-and-spike elimination. Reid's update, by contrast, benefits when nonzeroes fall more heavily in U than in L, and is well-suited to elimination by local minimization.

As noted previously, update techniques can also be designed to promote sparsity in the updated factors L_k and U_k. Reid's update in

particular is intended to preserve sparsity, and Gay has also incorporated Reid's ideas in Saunders' technique.

Sparse right-hand side vectors

The linear systems of the simplex method, $By = a$ and $B^T\pi = z$, usually have not only a sparse matrix but a very sparse right-hand side: a is a column of the sparse matrix A, and the pricing form z has one nonzero when the basis is feasible and k nonzeroes when there are k infeasibilities. FTRAN and BTRAN routines can take advantage of this additional sparsity to a certain extent, depending on how they access L and U.

For purposes of illustration, consider first a simple lower-triangular system $Lx = d$. If the nonzero elements of L are available sequentially by column, back-substitution is carried out as follows:

FTRANL:

REPEAT FOR j FROM 1 TO m:

SET $x_j = d_j/L_{jj}$

REPEAT FOR $L_{ij} \neq 0$, i FROM $j+1$ TO m:

SET $d_i = d_i - L_{ij}x_j$

At the jth pass through the main loop, if $d_j = 0$ then also $x_j = 0$ and the inner loop merely adds zero to various elements of d. Hence the jth pass is superfluous when $d_j = 0$. Moreover, if it happens that $d_1, \ldots, d_k$ are all zero, then the main loop does no work until pass k+1. A more efficient algorithm is thus as follows:

FTRANL:

1: SET $k = \min\{j : d_j \neq 0\}$; SET $x_j = 0$ for $j = 1, \ldots, k$

2: REPEAT FOR j FROM $k+1$ TO m:

IF $d_j = 0$: SET $x_j = 0$

ELSE: SET $x_j = d_j / L_{jj}$

REPEAT FOR $L_{ij} \neq 0$, i from $j+1$ TO m:

SET $d_i = d_i - L_{ij} x_j$.

Step 1 is especially valuable when $d_1, \ldots, d_k$ are known beforehand to be zero. In step 2, d tends to fill in with nonzeroes in each pass of the loop; but if L and d are both sparse then d should not fill in too quickly.

The situation is quite different if instead one must solve the upper-triangular system $L^T x = d$. If the nonzeroes of L are only available sequentially by column, then L^T is effectively available only by row, and back-substitution must be carried out as follows:

BTRANL:

REPEAT FOR j FROM m TO 1:

REPEAT FOR $L_{ij} \neq 0$, i FROM m to $j+1$:

SET $d_j = d_j - L_{ij} x_i$

SET $x_j = d_j / L_{jj}$

Here there is no advantage to knowing $d_j = 0$, since d_j is continually modified within the inner loop and x_j is not set until after the inner loop. The most one can say is that, if $d_m, \ldots, d_k$ are all zero, then $x_m, \ldots, x_k$ are also all zero and the main loop may be started with $j = k-1$.

For sparse elimination with updating the situation is somewhat more complex, involving not one L but a series of permuted L's. The conclusions are the same, however: if the lower-triangular factors of the basis are stored by column only--as they commonly are--then FTRANL can benefit from sparsity in the right-hand side to a much greater extent than BTRANL. Moreover, the same reasoning can be applied to U: if all or part of the upper-triangular factor is stored by column only, then BTRANU can exploit right-hand side sparsity much more than FTRANU.

In practice these differences have various consequences. At a typical iteration, the FTRAN and BTRAN operations are carried out once each, to solve systems that look like these:

TO SOLVE $By = a$:

FTRANL: $(P_0^T L_0)(P_1^T L_1) \cdot \cdots \cdot (P_k^T L_k)\ y^{(1)} = a$

BTRANU: $(U_k Q_k)\ y = y^{(1)}$

TO SOLVE $B^T \pi = z$:

FTRANU: $(Q_k^T U_k^T)\pi^{(1)} = z$

BTRANL: $(L_k^T P_k) \cdot \cdots \cdot (L_1^T P_1)(L_0^T P_0)\pi = \pi^{(1)}$

Hence sparsity of the right-hand side can be exploited in the following ways:

FTRANL can fully exploit the sparsity of a. A small additional advantage can be had if it is known that $(P_0 a)_1, \ldots, (P_0 a)_i$ are all zero for some i; this knowledge is not readily available in the general case, but it is often available from staircase methods to be described.

BTRANU can fully exploit any sparsity in $y^{(1)}$. Since $y^{(1)}$ is the solution vector from a sparse FTRANL, it may well be sparse itself.

FTRANU can exploit the considerable sparsity in z only if either U_k is available by row, or $(Q_k z)_1, \ldots, (Q_k z)_i$ are all zero from some i. In many cases it is possible to arrange that i is quite close to m [21]. Indeed, with some updating methods it can be guaranteed--provided the basis is feasible--that $(Q_k z)_1, \ldots, (Q_k z)_{m-1}$ are all zero, so that FTRANU may effectively be skipped.

BTRANL generally cannot benefit from sparsity in $\pi^{(1)}$. However, the update factors $L_1, \ldots, L_k$ are generally so simple in form that BTRANL handles them as efficiently as FTRANL. The significant extra work lies entirely in processing L_0^T.

Partial solutions

It is evident from the preceding analysis that the solution to $By = a$ or $B^T\pi = z$ is ultimately computed one element at at time, regardless of how L and U are stored. The vector y is produced by BTRANU in the order $(Q_k y)_m, \ldots, (Q_k y)_1$; likewise, the vector π is computed by BTRANL in the order $(P_0\pi)_m, \ldots, (P_0\pi)_1$.

BTRANL or BTRANU may therefore be terminated prematurely if only part of y or π needs to be computed. Such a partial solution has two potential uses in linear programming: when the rest of y is known to be zero, and when only a portion of π is required for pricing in the current iteration.

Nevertheless, in the general case there is little to be gained from trying to compute partial solutions, owing to the presence of permutations P_0 and Q_k: there is no efficient way to tell whether all remaining elements of $Q_k y$ are zero, or to predict which elements of $P_0 \pi$ will be needed. Section 4 will show, however, that partial solutions <u>can</u> offer an economy in solving staircase LPs, provided P_0 and Q_k are chosen to reflect the staircase structure.

3. SPARSE ELIMINATION OF STAIRCASE BASES

Two techniques for sparse elimination of staircase matrices were proposed in [15]: one adapts the bump-and-spike approach, while the other is a kind of local minimization. Either of these techniques may be applied to the staircase bases that arise from staircase LPs in the simplex method.

This section summarizes the direct effects--on speed, storage, and sparsity--of substituting staircase elimination techniques for standard ones in a simplex LP code. Section 4 then shows how these staircase techniques make possible additional efficiencies in the FTRAN and BTRAN routines.

Bump-and-spike techniques

The standard bump-and-spike technique [24,25] is a two-step procedure. First it determines the block-traiangular reduction of the basis B, an essentially unique permutation that puts B in block-triangular form with as many diagonal blocks ("bumps") as possible. Second, each diagonal block larger than 2×2 is further permuted by the Preassigned Pivot Procedure (P3), a heuristic that tries to make each block lower triangular except for a small number of "spike" columns that extend above the diagonal. Permuted in this way, B has a good structure for sparse Gaussian elimination: fill-in (creation of new nonzeroes during elimination) is confined to the spike columns, and pivots within a given bump cannot give rise to fill-in within other bumps.

A proposed staircase bump-and-spike technique [15] dispenses with block-triangular reduction, and uses instead the staircase form of the basis. The heuristic P3, adapted to handle blocks that are non-square or rank-deficient, is applied in turn to each of the diagonal blocks ($B_{\ell\ell}$) of the staircase. Thus the rows of period 1 are assigned to pivot first, followed by the rows of period 2, period 3, and so forth through period t. The columns are also generally pivoted in period order, but "interperiod spikes" from certain periods are pivoted in later periods in order to square off the oblong staircase blocks. Thus fill-in is confined to two kinds of spikes--intraperiod spikes found by P3, and interperiod spikes assigned to square off diagonal blocks--and pivots within a given period can only give rise to fill-in within spikes of the same period or within interperiod spikes of preceding periods. The balance constraints of Section 1 guarantee that this is a workable arrangement: the number of interperiod spikes need not be very large, and there are always enough interperiod spikes to square off every staircase block.

Computational experience [15] has shown that the standard and staircase bump-and-spike techniques are roughly comparable. They usually produce about the same number of spikes, and both yield a sparse factorization: the fill-in due to either technique is seldom more than twice the fill-in due to the other. However, each technique does appear to be superior in certain situations.

Standard bump-and-spike seems invariably better when all bumps are small and most are 1×1. P3 is then applied cheaply to a few blocks, whereas the staircase technique must still apply P3 to every diagonal

block of the staircase. The interperiod spikes of the staircase technique also tend to be larger than the spikes of the standard technique, and so the former fill in more: fill-in within L tends to be about the same, but the standard technique produces a notably sparser U. In addition, the standard technique is less prone to producing spikes that have unacceptable pivot elements, and so less time is wasted in "spike-swapping" during the elimination.

Staircase bump-and-spike has the advantage when there are one or two very large bumps that comprise half or more of the rows and columns of B. P3 becomes highly inefficient in processing these large bumps. Fill-in within U is comparable, while the staircase technique yields a sparser L. Moreover, the staircase technique produces substantially fewer spikes that have unacceptable pivots.

Storage requirements vary somewhat with the size of the largest block that must be processed, but are moderate in any case. Since a pivot order is fully chosen prior to elimination, storage required by the bump-and-spike heuristics may later be used to hold part of L and U.

Local-minimization techniques

Standard local-minimization techniques dynamically choose the kth pivot element from the remaining uneliminated matrix, $\beta^{(k)}$. The chosen pivot minimizes some "merit" function over all nonzero elements of $\beta^{(k)}$ that meet certain numerical tolerances. Practical merit functions are computed from two sets of values: $r_i^{(k)}$, the number of nonzeroes in row i of $\beta^{(k)}$, and $c_j^{(k)}$, the number of nonzeroes in column j of $\beta^{(k)}$.

Local minimization was first suggested by Markowitz [34], who proposed that the merit of element (i,j) be $(r_i^{(k)} - 1)(c_j^{(k)} - 1)$; no substantially better merit function has been found since.

Proposed staircase local-minimization techniques [15] differ by limiting the minimization to roughly one period of $\beta^{(k)}$ at a time. As a consequence both the rows and columns of B are pivoted in period order. It can also be shown that fill-in is limited to a small part of $\beta^{(k)}$--roughly two periods or less--while the remainder of $\beta^{(k)}$ is just the same as B.

Staircase local-minimization offers clear economies in both execution time and storage space. All of the work at the kth pivot--minimizing the merit function, updating $\beta^{(k)}$ to $\beta^{(k+1)}$, and updating $r_i^{(k)}$, $c_j^{(k)}$--is confined to the rows and columns of one or two periods, whereas the standard technique must deal with the entire $\beta^{(k)}$. Storage is required only for the part of $\beta^{(k)}$, also one or two periods, that differs from B.

For large problems of many periods, the differences in required storage may be immense. As a result, staircase local-minimization may be able to use simpler or more efficient storage strategies than standard local-minimization. During elimination by the standard technique the uneliminated $\beta^{(k)}$ shrinks while L and U grow; thus some sort of dynamic storage allocation is necessary when $\beta^{(k)}$, L and U are too large to be stored fully together. By contrast, under the staircase technique the active part of $\beta^{(k)}$ is small and fairly constant in size, and might well be kept in a fixed work area.

Standard local minimization does seem to usually produce a sparser L and U, as might be expected: it conducts its minimization over a much greater number of potential pivots. In the worst case in [15] the staircase technique produced about twice the fill-in (47% vs 22%); in some cases it did nearly as well, however, and in one it was distinctly better.

Comparison of techniques

Choice of a sparse-elimination technique cannot be separated from choice of an updating method (as explained previously), and both choices are sensitive to the nature and availability of storage. Consequently it is impossible to recommend one class of techniques--bump-and-spike or local-minimization--over the other categorically. Each may have its place in certain situations.

Indeed, the evidence of [15] suggests that every technique outlined in this section (standard and staircase bump-and-spike, standard and staircase local-minimization) offers the lowest fill-in for certain bases. Either of the staircase techniques should be acceptably fast, and all but the standard local-minimization have unproblematical storage requirements.

Staircase bump-and-spike techniques apply just as well to higher-order staircases. Staircase local-minimization might also be adapted to handle higher-order problems, but the extent of fill-in would be greater and hence the savings would be less.

4. SOLVING LINEAR SYSTEMS WITH STAIRCASE BASES

Both proposed staircase elimination techniques order their row pivots by period: all rows in period 1 are pivoted first, then all rows in period 2, and so forth. Staircase local-minimization also orders all column pivots by period, as does staircase bump-and-spike with the exception of certain columns (the interperiod spikes) that pivot after other columns of later periods.

This section describes how these staircase pivot orders can be taken advantage of to make the FTRAN and BTRAN routines more efficient. A partition of the L and U factors by period is first defined more formally, after which each solution routine--FTRANL, BRRANU, FTRANU, BTRANL--is taken up in turn.

Period partitions of the L and U factors

In the notation of Section 2, the basis B at an arbitrary iteration is factored as

$$B = (P_0^T L_0)(P_1^T L_1) \cdot \cdots \cdot (P_k^T L_k)(U_k Q_k)$$

In terms of this factorization and the staircase constraint matrix A, one may define the following indices for any period ℓ:

λ_ℓ first row of $P_0 B$ whose corresponding row of A is in period ℓ or later

μ_ℓ first column of BQ_k^T from period ℓ or later of A.

Necessarily $\lambda_\ell \leq \lambda_{\ell+1}$, $\mu_\ell \leq \mu_{\ell+1}$ for <u>any</u> factorization as above. Thus $\{\lambda_1, \ldots, \lambda_t\}$ and $\{\mu_1, \ldots, \mu_t\}$ partition the rows and columns, respectively, of $P_0BQ_k^T$ by period. Since the rows of $P_0BQ_k^T$ correspond to the rows of L_0, the λ's can also be thought of as partitioning L_0; analogously, the μ's partition U_k.

In general these partitions are not particularly useful, as the λ's and μ's all tend to be small. In an extreme case, for example, if the first row of P_0B is a period-t row then $\lambda_1 = \cdots = \lambda_t = 1$. It is thus necessary to show that the staircase pivoting techniques yield worthwhile partitions whose λ's and μ's are more or less evenly spread out.

Consider first a factorization with no updates, $P_0BQ_0^T = L_0U_0$. Certainly the staircase techniques, applied to the staircase structure that B inherits from A, yield good partitions. Either technique yields $\lambda_\ell = \sum_1^{\ell-1} m_i + 1$. For bump-and-spike $\mu_\ell \geq \lambda_\ell$, and $\mu_\ell = \lambda_\ell$ if there are no all-zero rows in $B_{\ell\ell}$; for local minimization, $\mu_\ell = \sum_1^{\ell-1} n_i + 1$.

The situation is slightly more complicated if, as suggested in Section 1, B is put in reduced standard staircase form before the staircase pivoting techniques are applied. Some rows of B that correspond to period-ℓ rows of A may then be pivoted as if they were in period ℓ-1. As a consequence, one can say only that $\sum_1^{\ell-2} m_i + 1 \leq \lambda_\ell \leq \sum_1^{\ell-1} m_i + 1$; the λ's may be smaller, and the λ-partition less regular. Nevertheless, the λ's are still well spaced and constitute a useful partition, particularly if the periods are small and numerous.

As B changes and the factorization is updated, L_0 and the λ-partition are unchanged. U_0 is updated to U_k, however, and in the process the μ-partition is altered. Specifically, all of the common update

methods have the following action: a column of BQ_{k-1}^T is deleted, and a new column is inserted at some point <u>after</u> the deleted column to produce BQ_k^T. The μ-partition up to the deleted column and after the inserted column is therefore unchanged; but if μ_ℓ is between the two columns then its value drops by 1. The μ-partition is thus slowly degraded. Degradation should not be severe, however, for large LPs with the usual 50-100 updates between refactorizations.

It may be concluded, then, that staircase pivot-selection techniques do yield λ's and μ's that constitute non-trivial partitions of L and U by period.

<u>Staircase FTRANL</u>

At each iteration FTRANL starts by solving a system like $(P_0^T L_0)x = a$, or equivalently $L_0 x = P_0 a$, where a is a column of A. If a is from period ℓ, then it is zero on rows of periods 1 through $\ell-1$. Consequently,

$$(P_0 a)_i = 0 , \qquad i = 1,\ldots,\lambda_\ell - 1$$

and the main loop of the FTRANL routine may begin at index λ_ℓ as explained in Section 2.

In short, when FTRANL transforms a period-ℓ column it can start at the ℓth period in L_0, rather than at the beginning. The resultant savings will be small, however, since FTRANL already handles right-hand side zeroes efficiently.

Further savings might be possible if one kept track of upper-sub-staircases of B_0, as described in Section 1. The idea is as follows:

if B_0 has an upper-sub-staircase in periods 1 through ℓ, and if a lies in period ℓ or earlier, then the solution x of $(P_0^T L_0)x = a$ is zero in periods $\ell+1$ and later. Thus the main loop of FTRANL may be terminated prematurely. As a practical matter, however, the logic of such a scheme is fairly complex, and computational experiments [15] have shown only a moderate number of upper-sub-staircases; so the potential savings are probably not worth the trouble.

Staircase BTRANU

At each iteration BTRANU solves a system like $(U_k Q_k)y = x$, where x is a solution vector from FTRANL. Since FTRANL has solved with $L_0, L_1, \ldots, L_k$, there is no telling where zeroes may be in x. Hence BTRANU cannot benefit specially from a sparse right-hand side.

A small saving is possible, however, if the location of (lower) square sub-staircases in B is known. Suppose that the linear system at hand is $By = a$, that a is from period j, and that B has a sub-staircase at period $\ell < j$ (that is, $\sum_1^\ell m_i = \sum_1^\ell n_i$). Then the system can be partitioned as

$$\left[\begin{array}{c|c} B^{(11)} & 0 \\ \hline B^{(21)} & B^{(22)} \end{array}\right] \left[\begin{array}{c} y^{(1)} \\ \hline y^{(2)} \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline a^{(2)} \end{array}\right]$$

where $B^{(11)}$ and $B^{(22)}$ are the square sub-staircases. Clearly the solution must have $y^{(1)} = 0$, $y^{(1)}$ being just the part of y that corresponds to the columns of B in periods 1 through ℓ.

Now if $By = a$ is written instead as $(BQ_k^T)(Q_k y) = a$, the preceding statement is equivalent to the following: an element of $Q_k y$ will be zero if it corresponds to a column of BQ_k^T in periods $1,\ldots,\ell$. That is,

$$(Q_k y)_i = 0, \qquad i = 1,\ldots,\mu_\ell - 1$$

Thus the main loop of BTRANU, which computes $(Q_k y)_i$, $i = m,\ldots, 1$, can stop after the μ_ℓ-th pass; the remainder of the solution is zero.

Staircase FTRANU

FTRANU solves at each iteration a system like $(U_k Q_k)^T x = z$, or $U_k^T x = Q_k z$, where z is a pricing form chosen in one of several ways (see Section 2). Usually most of z is zero, and often it can be determined that z is zero in all columns of the first ℓ periods of the basis; during Phase I of the simplex method, for example, this would occur if all basic variables of the first ℓ periods were feasible. It would then follow that

$$(Q_k z)_i = 0, \qquad i = 1,\ldots,\mu_\ell - 1$$

and the main loop of FTRANU could begin at μ_ℓ as explained in Section 2.

This result is analogous to the one for FTRANL above: when FTRANU transforms a z that is zero prior to period ℓ, it can start at the ℓth period in U_k rather than at the beginning. However, the potential savings are greater since--if U_k is stored only by column--FTRANU cannot normally benefit from sparsity in z. In practice the savings depend on how U is actually stored and on how z is handled.

Staircase BTRANL

BTRANL produces a vector π that is employed in "pricing" nonbasic columns of A; specifically, each iteration computes numerous inner products $\pi^T a$ with columns a. If a is from period ℓ then it is zero except on rows of periods ℓ and $\ell+1$, and so only the elements of π that correspond to these periods are needed to form $\pi^T a$. Since the simplex method seldom considers all nonbasic columns at one iteration, it can be arranged that only certain periods of π are needed. (See [16] for a more extensive explanation.)

Assume, therefore, that at the current iteration one only needs elements of π corresponding to rows of periods ℓ and later. The vector π is the solution of $B^T\pi = z$, or $(P_0B)^T(P_0\pi) = z$. Thus, equivalently, one needs only elements of $P_0\pi$ that correspond to rows of P_0B in periods ℓ and later. It will suffice, therefore, to compute $(P_0\pi)_i$, $i = \lambda_\ell, \ldots, m$.

BTRANL actually produces the elements of π by solving $(P_0^T L_0)^T\pi = x$, or $L_0^T(P_0\pi) = x$, where x has been obtained from preceding transformations of z in FTRANU and BTRANL. Each pass through BTRANL computes another element of $P_0\pi$, in reverse order: $(P_0\pi)_m, \ldots, (P_0\pi)_1$. Thus to compute the desired part of π one need only run BTRANL through the λ_ℓth pass of the main loop; the remainder may be skipped.

The potential savings in this instance are considerable. Using one of the partial-pricing schemes of [16] substantial amounts of computation may be avoided, on the average, at each iteration. This is especially important as BTRANL is one of the less efficient transformations, being unable to take advantage of right-hand side sparsity when L_0 is stored in the usual columnwise fashion.

5. COMPUTATIONAL EXPERIENCE

This section reports on initial computational experiments with some of the preceding ideas. The results indicate that staircase adaptation of the simplex method does make a significant difference: generally much less time is spent in certain routines, while more time is spent in others. Overall the staircase runs were measurably faster, and in one case the savings were quite substantial. Moreover, it appears there is still room for improvement in subsequent implementations.

For the test runs an existing LP code, MINOS [38,48], was modified to recognize staircase structure and to apply optionally the staircase techniques of Sections 3 and 4. Each test LP could then be solved twice --once with the staircase features turned off, once with them on--and the results could be meaningfully compared. Details of the test code and the experimental setup are given in Appendix B.

MINOS employs a bump-and-spike factorization with Saunders' updating technique. Consequently the staircase bump-and-spike technique was implemented in the test version, and all test results bear directly only upon bump-and-spike methods. Nevertheless, from certain results one may make quite favorable speculations about the expected performance of staircase local-minimization techniques, as described further below.

To keep the presentation compact, only short tables of results are presented in this section. Graphs of more extensive test data are collected in Appendix C.

Overall results

Seven medium-to-large-scale linear programs were used in the tests. All are from applications, and are of dissimilar structures (aside from being staircase). Their dimensions are as follows:

	PERIODS	ROWS	COLUMNS	NONZERO COEFFICIENTS	ITERATIONS TO SOLVE FROM SLACK START
SCAGR25	25	472	500	2208	1058
SCRS8	16	491	1169	4106	862
SCSD8	39	398	2750	11,349	2047
SCFXM2	8	661	914	5466	1012
SCTAP2	10	1101	1880	13,815	1174
PILOT	9	723	2789	9291	>2000
BP1	6	822	1571	11,414	>2000

For the sake of economy, PILOT and BP1 were tested on runs of 1000 and 750 iterations, respectively, starting from advanced bases. The rest were run to optimality from an all-slack start. Additional information about the test LPs is collected in Appendix A, and Appendix B explains in more detail how they were solved.

Raw results from the test runs, standardized to seconds per 1000 iterations, were as follows:

	TOTAL TIME		
	STANDARD	STAIRCASE	% CHANGE
SCAGR25	29.7	27.9	- 6%
SCRS8	33.9	31.5	- 7%
SCSD8	43.2	37.8	-13%
SCFXM2	43.4	42.2	- 3%
SCTAP2	67.2	67.1	0%
PILOT	155.7	106.4	-32%
BP1	181.8	189.7	+ 4%

Savings were substantial for PILOT, and respectable for SCSD8. For the others the gross difference between the standard and staircase techniques was small, though the latter performed worse only on BP1.

It is misleading to consider only these totals, however. When the times are broken down by function--as in the first set of graphs in Appendix C--it can be seen that gains in some areas tend to be offset by losses in others. The staircase version has an edge in simplex pricing and pivoting, while it is usually slightly behind in updating the LU factorization; it ranges from much faster to somewhat slower in pivot selection for Gaussian elimination, but is almost always slower in computing the L and U factors. Miscellaneous routines consume a good 10-20% of the time, much of which could be saved in practical (rather than test) circumstances.

Thus much more is to be learned by examining the times of individual routines and functions. The following subsections consider first the simplex-iteration routines, and then the LU-factorization ones.

Iterating routines

The simplex method spends a majority of its time in tasks that are repeated at each iteration: choosing a column to enter the basis (pricing), determining which column leaves the basis (pivoting), and revising the basis factorization accordingly (updating). The LP code's "iterating" routines carry out these tasks.

For the test problems, total time spent in the iterating routines --again, normalized to seconds per thousand iterations--was as follows:

	ITERATING TIME		
	STANDARD	STAIRCASE	% CHANGE
SCAGR25	24.6	22.2	-10%
SCRS8	28.1	23.8	-15%
SCSD8	34.2	30.5	-11%
SCFXM2	33.2	32.5	- 2%
SCTAP2	56.9	54.3	- 5%
PILOT	108.0	86.3	-20%
BP1	136.6	146.1	+ 7%

Here the results are somewhat more striking, four of the seven showing savings of 10-20%.

Again more can be learned from a further breakdown of the times, given by the second set of graphs in Appendix C. The greatest difference by far is in BTRANL, which is significantly faster with the staircase version in every instance. There is a corresponding, but smaller, efficiency in FTRANL. The figures for these two routines are as follows:

	FTRANL			BTRANL		
	STD	STAIR	% CHNG	STD	STAIR	% CHNG
SCAGR25	2.7	1.9	-29%	6.7	3.5	-48%
SCRS8	2.4	1.5	-36%	5.7	3.4	-41%
SCSD8	3.9	2.9	-25%	8.2	4.7	-42%
SCFXM2	2.6	1.9	-28%	7.8	5.4	-32%
SCTAP2	3.3	2.6	-21%	9.2	6.6	-28%
PILOT	13.0	8.0	-38%	22.9	12.7	-45%
BP1	14.8	12.6	-15%	32.5	26.9	-17%

Roughly there is a 30-50% saving in BTRANL, and a 20-40% saving in FTRANL.

There is a small but noticeable tendency of the staircase version to run slower in BTRANU and FTRANU. Most likely this behavior is a consequence of the LU factorization: the staircase bump-and-spike pivot order tends to yield a denser U.

Some of the difference in BTRAN and FTRAN timings should be due to the methods of Section 4. The efficacy of these methods cannot be told from the above data, however, since the same timings are sensitive to differences in L and U density. Consequently a separate set of runs was made, employing the staircase LU factorization but not the Section 4 enhancements. The differences were as follows:

	TIME SAVED BY EFFICIENCIES IN FTRAN, BTRAN (SECTION 4)	% OF TOTAL TIME
SCAGR25	4.9	15%
SCRS8	4.1	12%
SCSD8	5.2	12%
SCFXM2	4.4	9%
SCTAP2	4.4	6%
PILOT	13.4	11%
BP1	3.5	2%

Thus the efficiencies in FTRAN and BTRAN cut total running times 9-15% in most cases; the savings would be more pronounced as a percentage of iterating time only. Predictably, LPs of many periods tended to show the greatest differences.

Comparable savings should be realized if staircase bump-and-spike pivot selection is replaced by staircase local minimization, since the methods of Section 4 apply equally well to either. Hence local minimization may well be superior for LPs such as SCAGR25 and SCFXM2 whose staircase factorizations--as reported in [15]--are notably denser under bump-and-spike.

The one sour note in the three tables above is BP1, on which the staircase iterating routines seem to perform rather poorly. On closer examination, however, this is not entirely surprising, as BP1 differs significantly from the other LPs. Whereas the others are first-order staircases (or, in the case of PILOT, very nearly first-order), BP1 has a large

number of nonzeroes below the staircase; its form is in fact closer to dual-angular. BP1's bases consequently tend to be unbalanced. Hence the staircase technique produces considerably more spikes, and a much denser U factor. The result: much more time spent in FTRANU and BTRANU, offsetting any gains in FTRANL and BTRANL.

It thus appears that a good staircase form is essential to success of the staircase techniques. BP1's staircase arrangement was deduced from fairly scant information, and is evidently inadequate. A better staircase form may exist, but a better knowledge of the underlying model may be necessary to find it.

Factorizing routines

At intervals of typically 50-100 iterations a fresh factorization of the basis is computed by a separate set of routines. For bump-and-spike techniques, these "factorizing" routines fall into two classes: ones that select a pivot order, and ones that compute the L and U factors.

For the test problems, total time in factorizing routines--normalized to seconds per 10 refactorizations--was as follows:

	FACTORIZING TIME		
	STANDARD	STAIRCASE	% CHANGE
SCAGR25	1.4	1.6	+15%
SCRS8	1.1	1.4	+22%
SCSD8	2.7	1.6	-39%
SCFXM2	1.9	2.8	+47%
SCTAP2	1.7	3.0	+80%
PILOT	32.8	9.7	-70%
BP1	27.9	26.1	- 6%

The outcomes appear to vary wildly. However, they are the consequence of a few simple patterns which are revealed by looking at the pivot-selection routines and LU-computation routines separately, with reference to the third set of graphs in Appendix C.

Pivot selection involves a routine for the P3 heuristic, a block-triangularization routine (for the standard technique only), and main routines to call these and record the selected pivots. The staircase technique's main routine seems to run usually somewhat longer, probably because it is more complicated. The others' times are summarized below:

	STANDARD			STAIRCASE	MEDIAN SIZE,
	P3	BLK Δ	TOTAL	P3	LARGEST BUMP
SCAGR25	0.4	0.2	0.6	0.2	45
SCRS8	0.2	0.2	0.4	0.2	28
SCSD8	1.1	0.4	1.5	0.2	114
SCFXM2	0.2	0.5	0.7	0.8	36
SCTAP2	0.0	0.5	0.5	0.7	1
PILOT	20.4	1.0	21.4	2.4	533
BP1	13.1	2.0	15.1	3.8	408

The behavior of P3 is clearly critical. When bumps are small P3 is quite fast; but it begins to slow down whem bump size passes 100, and it is extremely inefficient on bumps of size 400 or 500. PILOT, the worst case here, spends 16% of its total running time in P3 alone! By extrapolation, it seems likely that P3 will be prohibitively slow for larger bumps. Thus a staircase bump-and-spike technique (or else an efficient local-minimization technique) may be essential for larger versions of models like SCSD8 and PILOT.

The main LU computation routines employ FTRANL and BTRANL as subroutines: FTRANL solves for the next column of L and U (as described in Section 2); BTRANL solves for row k of $\beta^{(k)}$ when a column interchange ("spike swap") is necessitated by an unacceptable pivot element. The test problems gave the following results (where SWAPS is the maximum number of swapped spikes per factorization):

	STANDARD LU				STAIRCASE LU			
	MAIN	FTRAN	BTRAN	SWAPS	MAIN	FTRAN	BTRAN	SWAPS
SCAGR25	0.2	0.0	0.0	3	0.6	0.1	0.1	20
SCRS8	0.2	0.0	0.0	1	0.4	0.1	0.1	11
SCSD8	0.4	0.1	0.0	6	0.5	0.1	0.1	11
SCFXM2	0.5	0.1	0.0	2	1.0	0.2	0.1	8
SCTAP2	0.2	0.0	0.0	0	0.7	0.1	0.4	19
PILOT	3.3	3.8	2.4	27	3.2	1.8	0.8	16
BP1	3.7	3.8	2.4	28	6.4	5.8	7.2	49

Predictably, the times are sensitive to the numbers of spike swaps; each swap requires another BTRANL and FTRANL, plus extra work in the main routine. Experience with PILOT and other LPs [15] suggests that the staircase pivot order may generally require fewer swaps when the bumps are big (as for PILOT) and the staircase is well-balanced (unlike BP1's). The other test LPs have smaller bumps and require fewer swaps with the standard pivot order.

Again the data suggest that staircase local-minimization techniques might be preferable for the small-bump staircase LPs. An efficient implementation of local minimization [12,45] incurs only a small extra cost in rejecting any unaccepatably small pivot element.

Comparison with a commercial code

The PILOT model was frequently solved--on the same computer as used for the above tests--by a commercially-marketed machine-language LP code, MPS III [37]. These runs employed the WHIZARD simplex routine of MPS III, which incorporates a bump-and-spike factorization scheme. Various system parameters were set from experience to yield fast PILOT runs.

For comparison, WHIZARD was run 1000 iterations from the same starting basis as used above with MINOS. The running times were as follows:

MINOS, standard pivot selection	155.7 sec
MPS III/WHIZARD	114.7 sec
MINOS, staircase pivot selection	106.4 sec.

MINOS did require considerably more storage, primarily because its storage scheme for the U factor could not efficiently accommodate a large number of spikes. U could probably be stored more compactly, however, without significant effect upon the MINOS timings.

Nothing very definite can be inferred from these figures, since MINOS and MPS III differ in many ways; moreover, the internal structure of the latter is largely unknown, as is the case with many commercial codes. Nevertheless, it is gratifying that MINOS--which is written in FORTRAN and intended more as a test code--can compete with a supposedly fast LP system. At the least, one may conclude that the timings throughout this section are probably quite realistic. And the superiority of staircase MINOS to MPS III for PILOT suggests that, for at least some large staircase problems, the techniques of this paper will offer significant savings.

APPENDIX A: TEST PROBLEMS

The linear programs used in the computational experiments of Section 5 are described in greater detail below. The tabular summarizes for each LP are largely self-explanatory, but a few general notes are appropriate:

All statistics except OBJ ELEMS refer only to the staircase constraint matrix, excluding the objective row and right-hand side. In each case the constraint matrix, A, has been put in reduced standard form; DIAGONAL BLOCKS refers to the staircase blocks $A_{\ell\ell}$, OFF-DIAGONAL BLOCKS to the blocks $\hat{A}_{\ell+1,\ell}$, and SUB-STAIR BLOCKS (when present) to the blocks $A_{\ell+2,\ell}, \cdots, A_{t\ell}$.

Variables (columns) are implicitly constrained only to be non-negative, unless there is an indication to the contrary. BOUNDED implies implicit lower and upper bounds, FIXED implies fixture at a given value, and FREE implies no implicit constraints.

MAX ELEM and MIN ELEM are the largest and smallest magnitudes of elements in A; LARGEST COL RATIO is the greatest ratio of magnitudes of elements in the same column of A. Where values are given BEFORE SCALING and AFTER SCALING, all tests were conducted with A scaled as described in Appendix B. Otherwise NO SCALING is indicated.

SCAGR25

Test problem received from James K. Ho, Brookhaven National Laboratory, Upton, N.Y.; source not documented.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1	18	20	45	13%	8	7	17	30%	19
2-24	19	20	46	12%	8	7	17	30%	19
25	16	20	43	13%					19
			1146	12%			408	30%	475

GRAND TOTALS

ROWS	471	(300 EQUALITIES, 171 INEQUALITIES)
COLS	500	
ELEMS	1554	
DENS	0.7%	

COEFFICIENTS	NO SCALING
MAX ELEM	1.3
MIN ELEM	2.0×10^{-1}
LARGEST COL RATIO	1.9×10^{-1}

SCRS8

Derived from a model of the United States' options for a transition from oil and gas to synthetic fuels; documented in [27,33].

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1	28	37	65	6%	25	22	29	5%	18
2	28	38	69	6%	25	22	29	5%	19
3-5	31	76	181	8%	25	22	29	5%	55
6-8	32	79	192	8%	25	22	29	5%	58
9	31	79	189	8%	25	22	29	5%	58
10-12	31	80	190	8%	25	22	29	5%	59
13-15	30	80	186	8%	25	22	29	5%	59
16	31	70	177	8%					59
			2747	8%			435	5%	847

GRAND TOTALS

ROWS	490	(384 EQUALITIES, 106 INEQUALITIES)
COLS	1169	
ELEMS	3182	
DENS	0.6%	

COEFFICIENTS	BEFORE SCALING	AFTER SCALING
MAX ELEM	3.9×10^{2}	4.0
MIN ELEM	1.0×10^{-3}	2.5×10^{-1}
LARGEST COL RATIO	4.5×10^{3}	1.6×10^{1}

SCSD8

A multi-stage structural design problem, documented in [26]. This is the only staircase test problem for this paper in which the stages do not represent periods of time.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1-38	10	70	130	19%	10	50	90	18%	70
39	17	90	224	15%					90
			5164	18%			3420	18%	2750

GRAND TOTALS

ROWS	397	(ALL EQUALITIES)
COLS	2750	
ELEMS	8584	
DENS	0.8%	

COEFFICIENTS	NO SCALING
MAX ELEM	1.0
MIN ELEM	2.4×10^{-1}
LARGEST COL RATIO	4.0

SCFXM2

Test problem received from James K. Ho, Brookhaven National Laboratory, Upton, New York; source not documented.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1	92	114	679	6%	9	57	61	12%	13
2	82	99	434	5%	9	35	35	11%	4
3	66	126	300	4%	5	33	33	20%	1
4	90	118	1047	10%	5	5	5	20%	5
5	92	114	679	6%	9	57	61	12%	13
6	82	99	434	5%	9	35	35	11%	4
7	66	126	300	4%	5	33	33	20%	1
8	90	118	1047	10%					5
			4920	7%			263	13%	46

GRAND TOTALS

ROWS	660	(374 EQUALITIES, 286 INEQUALITIES)
COLS	914	
ELEMS	5183	
DENS	0.9%	

COEFFICIENTS	BEFORE SCALING	AFTER SCALING
MAX ELEM	1.3×10^{2}	1.1×10^{1}
MIN ELEM	5.0×10^{-4}	8.7×10^{-2}
LARGEST COL RATIO	1.3×10^{5}	1.3×10^{2}

SCTAP2

A dynamic traffic assignment problem, documented in [28]. The LP has 11 objective rows; the objective named OBJZZZZZ was used in all tests. Statistics below omit the other ten objectives.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1-9	109	188	423	2%	62	138	276	3%	141
10	109	188	423	2%					141
			4230	2%			2484	3%	1410

GRAND TOTALS

ROWS	1090	(470 EQUALITIES, 620 INEQUALITIES)
COLS	1880	
ELEMS	6714	
DENS	0.3%	

COEFFICIENTS	NO SCALING
MAX ELEM	8.0×10^{1}
MIN ELEM	1.0
LARGEST COL RATIO	8.0×10^{1}

PILOT

Derived from a welfare equilibrium model of the United States' energy supply, energy demand, and economic growth: seeks maximum aggregate consumer welfare subject to competitive market equilibrium. The LP was supplied by the PILOT modeling project, Systems Optimization Laboratory, Department of Operations Research, Stanford University; it is documented in [40].

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				SUB-STAIR BLOCKS		OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS	DENS	ELEMS
1	84	343	686	2%	31	74	105	5%	18	0%	10
2	90	345	1079	3%	34	76	111	4%	8	0%	10
3	90	343	1073	3%	34	74	109	4%	5	0%	10
4	90	343	1073	3%	34	74	109	4%	5	0%	10
5	90	343	1073	3%	34	74	109	4%	5	0%	10
6	90	343	1073	3%	34	74	109	4%	3	0%	10
7	90	343	1073	3%	32	74	107	5%	1	0%	10
8	87	341	1060	4%	4	19	19	25%			10
9	11	45	113	23%							12
			8303	3%			778	4%	45	0%	92

GRAND TOTALS

ROWS	722	(583 EQUALITIES, 139 INEQUALITIES)
COLS	2789	(80 FREE, 296 BOUNDED, 79 FIXED)
ELEMS	9126	
DENS	0.5%	

COEFFICIENTS	BEFORE SCALING	AFTER SCALING
MAX ELEM	4.8×10^{4}	2.0×10^{1}
MIN ELEM	1.4×10^{-4}	4.9×10^{-2}
LARGEST COL RATIO	7.0×10^{6}	4.2×10^{2}

BP1

Developed by British Petroleum, London; supplied via the Systems Optimization Laboratory, Department of Operations Research, Stanford University.

This LP is approximately dual-angular, with 6 main diagonal blocks and about 400 coupling variables. For the experiments described in this paper it was treated as a 6-period, 5th-order staircase problem.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				SUB-STAIR BLOCKS		OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS	DENS	ELEMS
1	111	227	1400	6%	3	60	3	2%	163	0%	138
2	151	353	2175	4%	62	108	112	2%	142	0%	149
3	113	321	964	3%	92	232	346	2%	494	1%	270
4	170	295	2178	4%	51	14	11	2%	4	0%	74
5	134	198	1315	5%	111	2	2	1%			40
6	142	177	1091	4%							56
			9123	4%			474	2%	803	0%	727

GRAND TOTALS

ROWS	821	(516 EQUALITIES, 305 INEQUALITIES)
COLS	1571	
ELEMS	10400	
DENS	0.8%	

COEFFICIENTS	BEFORE SCALING	AFTER SCALING
MAX ELEM	2.4×10^{2}	1.3×10^{1}
MIN ELEM	2.0×10^{-4}	7.6×10^{-2}
LARGEST COL RATIO	1.7×10^{5}	1.7×10^{2}

APPENDIX B: DETAILS OF COMPUTATIONAL TESTS

Computing environment

All computational experiments were performed on the Triplex system [49] at the Stanford Linear Accelerator Center, Stanford University. The Triplex comprises three computers linked together: one IBM 360/91, and two IBM 370/168s. Runs were submitted as batch jobs in a virtual-machine environment, under the control of IBM systems OS/VS2, OS/MVT and ASP.

Test runs employed a specially-modified set of linear-programming routines from the MINOS system [38,48]. MINOS is written in standard FORTRAN. For timed runs, MINOS was compiled with the IBM FORTRAN IV (H extended, enhanced) compiler, version 1.1.0, at optimization level 3 [30].

Timings

All running-time statistics are based on "CPU second" totals for individual job steps as reported by the operating system. To promote consistency all timed jobs were run on the Triplex computer designated "system A," and jobs whose timings would be compared were run at about the same time. Informal experiments indicated roughly a 1% variation in timings due to varying system loads.

More detailed timings employed PROGLOOK [31], which takes frequent samples of a running program to estimate the proportion of time spent in each subroutine. To determine the actual time in seconds for each subroutine, every timed job was run twice--once without PROGLOOK to measure total CPU seconds, and once with PROGLOOK to estimate each subroutine's proportion of the total. PROGLOOK estimates were based on at least 2300 samples per job.

MINOS linear-programming environment

MINOS was set up for test runs according to the defaults indicated in [38], with the exception of the items listed below.

Scaling. Problems noted as "scaled"in Appendix A were subjected to the following geometric-mean scaling (where A denotes the matrix of constraint coefficients, not including the objective or right-hand side):

1: Compute $\rho_0 = \max|A_{i_1 j}/A_{i_2 j}|$, $A_{i_2 j} \neq 0$.

2: Divide each row i of A, and its corresponding right-hand side value, by $[(\min_j|A_{ij}|)(\max_j|A_{ij}|)]^{1/2}$, taking the minimum over all $A_{ij} \neq 0$.

3: Divide each column j of A, and its corresponding coefficient in the objective, by $[(\min_i|A_{ij}|)(\max_i|A_{ij}|)]^{1/2}$, taking the minimum over all $A_{ij} \neq 0$.

4: Compute $\rho = \max|A_{i_1 j}/A_{i_2 j}|$, $A_{i_2 j} \neq 0$.

This procedure was repeated as many times as possible until, at step 4, ρ was at least 90% of ρ_0. (In other words, scaling continued as long as it reduced ρ, the greatest ratio of two elements in the same column, by more than 10%.)

Starting basis. All LPs except PILOT and BP1 were solved with crash option 0 of MINOS: the initial basis was composed entirely of unit vectors, and all nonbasic variables were placed at zero. PILOT and BP1 were run from initial bases that had been reached and saved in previous MINOS runs.

Termination. All LPs except PILOT and BP1 were run until an optimal solution was found. PILOT and BP1 were run for 1000 and 750 iterations, respectively.

Pricing. Except for SCTAP2, the partial-pricing scheme of MINOS was employed--with one important change: the arbitrary partitioning of the columns normally defined by MINOS for partial pricing was replaced by the natural staircase partition. Thus the periods of the staircase were priced one at a time in a cyclic fashion.

Pricing for SCTAP2 was similar except that the incoming column was chosen from the latest possible period. (This choice was known to produce a relatively small number of iterations from an all-unit-vector start.)

Refactorization frequency. MINOS was instructed to refactorize the basis (by performing a fresh Gaussian elimination) every 50 iterations, except for BP1 (every 75) and PILOT (every 90).

Tolerances. The "LU ROW TOL" for MINOS was set to 10^{-4}. All other tolerances were left at their default values.

Modifications to MINOS

All runs described in this paper were made with a special test version of MINOS. This version retained MINOS' routines for standard bump-and-spike elimination, and added new routines to implement a version of staircase bump-and-spike elimination. Routines for solving linear systems were also modified to take advantage of the staircase pivot order. Control routines were adjusted appropriately.

New subroutines in the test version are described briefly as follows:

SP3--an adaptation of the P3 heuristic to find a bump-and-spike structure in non-square or rank-deficient blocks, as proposed in [15]. This routine is a modification of the MINOS subroutine P3.

SP4--main routine for the staircase bump-and-spike pivot-selection technique of [15]; sorts the staircase basis into reduced form, and calls SP3 once for each diagonal block.

DSPSPK--spike-display routine; prints a graphical summary of the basis bump-and-spike structure found by P4 (for the standard technique) or SP4 (for the staircase technique).

STAIR--a staircase analyzer. Given an initial partition of the rows by period, this routine permutes the constraint matrix to a reduced standard staircase form and stores the staircase partitions in arrays that are read by subsequent routines. STAIR is called once at the beginning of every run.

SCALE--implementation of the geometric-mean scaling scheme described above; called optionally at the beginning of a run.

UPDBAL--updating routine for cumulative-balance counts: after each iteration, revises an array that records the cumulative excess of columns over rows at each period of the staircase basis. (This array is used to find square sub-staircases.)

In addition the test version incorporates the following substantial modifications to MINOS subroutines:

FACTOR efficiently handles a pivot order from either the standard or staircase technique, and finds the partitions λ_ℓ and μ_ℓ (defined in Section 4) for the staircase technique.

FTRANL, BTRANL, FTRANU and BTRANU incorporate the ideas of Section 4 in a uniform way. FTRANL and FTRANU can begin at a specified L or U transformation, and BTRANL and BTRANU can stop at a specified transformation. BTRANL can also be restarted at a point where it previously stopped.

LPITN determines a starting point for FTRANL and a stopping point for BTRANU when the staircase technique is used.

SETPI, for the staircase technique, determines a starting point for FTRANU and a stopping point for BTRANL when it is first called at an iteration. When subsequently called at the same iteration it determines restarting and stopping points for BTRANL.

PRICE incorporates the staircase-oriented partial-pricing methods described in the preceding subsection of this appendix. When these methods are used with the staircase factorization technique, PRICE also keeps track of how much of the price vector it requires, and calls SETPI accordingly.

SPECS2 determines whether the standard or staircase technique will be used in a particular run, according to instructions in the SPECS input file.

Other subroutines were modified as necessary to accommodate these changes.

MPS III linear programming environment

For purposes of comparison the PILOT test problem was also run on the MPS III system [37], as explained in Section 5.

The MPS III run employed the WHIZARD linear-programming routines of version 8915 of MPS III. The run used the same starting basis as the MINOS runs for PILOT, and was terminated after 1000 iterations like the MINOS runs. Exact CPU timings were 0.56 seconds in the compiler step and 114.18 seconds in the executor step.

The control program for the MPS III run was as follows:

```
            PROGRAM
            INITIALZ
            XPROC = XPROC + 6000
            XCLOCKSW = 0
            XINVERT = 1
            XFREQINV = 90
            XFREQLGO = 1
            XFREQ1 = 1000
            MVADR (XDOFREQ1, TIME)
            MOVE (XDATA, 'PILOT.WE')
            CONVERT ('FILE','INPUT')
            SETUP ('BOUND', 'BOUND', 'MAX', 'SCALE')
            MOVE (XOBJ, 'OBJ')
            MOVE (XRHS,'RHSIDE')
            INSERT ('FILE','PUNCH1')
WHIZFREQ    DC (250)
WHIZSCAL    DC (4)
            WHIZARD('FREQ', WHIZFREQ, 'SCALE', WHIZSCAL)
TIME        PUNCH ('FILE', 'PUNCH1')
            EXIT
            PEND
```

APPENDIX C: TIMINGS

The bar graphs below summarize timings of the MINOS test runs for this paper. Details of the test runs and timing procedures are in Appendix B; individual MINOS subroutines are documented in Appendix B and in [48].

Graphs are presented in three groups. The first group shows time in all routines, the second shows time in iterating routines only, and the third shows time in factorizing routines only. Within each group the format is the same: the first graph compares totals for all seven test problems, and seven succeeding graphs--one for each test problem--break the times down into various subtotals.

All graphs show a pair of bars for each total or subtotal. The top bar is for the run that used standard bump-and-spike elimination on the basis; the bottom bar is for the run that used staircase bump-and-spike elimination and the related techniques described in this paper.

Total time

The FORTRAN subroutines of MINOS are classified below as follows:

PRICE routines choose a nonbasic variable to enter the basis; they include FORMC, PRICE, SETPI and FTRANU, and BTRANL when called from SETPI.

PIVOT routines choose a variable to leave the basis; they include LPITN and CHUZR, and FTRANL, BTRANU and UNPACK when called from LPITN.

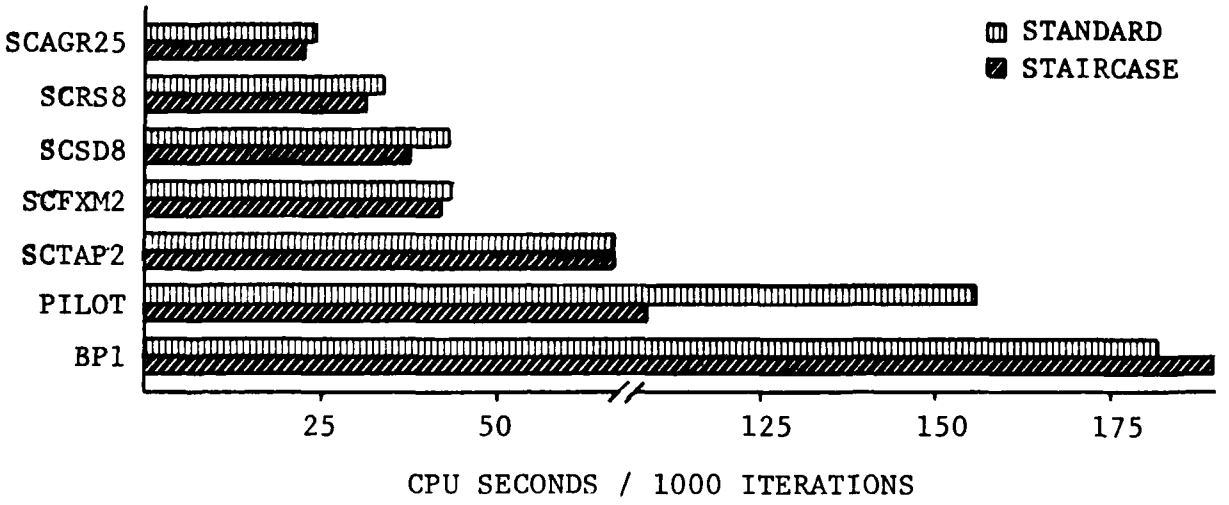
UPDATE refers to the subroutine MODLU, which updates the LU factorization of the basis at the end of each iteration.

PERM routines permute the basis of a bump-and-spike structure. For the standard method they include P4, P3, TRANSVL, BUMPS and MKLIST; for the staircase method they are SP4, SP3 and MKLIST.

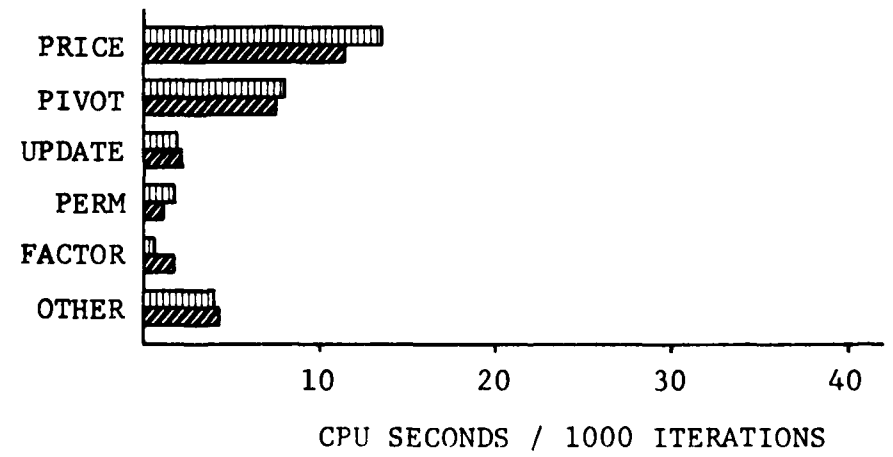
FACTOR routines compute an LU factorization of the basis; they include FACTOR and PACKLU, and FTRANL, BTRANL and UNPACK when called from FACTOR.

OTHER routines include all other MINOS subroutines, and utility routines inserted by the FORTRAN compiler. Other MINOS routines comprise DRIVER and routines it uses (BTRANU, FTRANL, ITEROP, SETX, STATE, UNPACK, UPDBAL), INVERT and routines it uses (BTRANU, DSPSPK, FTRANL, SETX), and various routines called once only at the beginning or end of the run (CRASH, GO, HASH, INITLZ, LOADB, MINOS, MOVE, MPS, MPSIN, NMSRCH, SAVEB, SCALE, SOLN, SOLPRT, SPECS, SPECS2, STAIRS). FORTRAN routines for input and output registered significantly (3-10% of total) in the timings; the volume of input was very small, so these routines probably did most of their work in producing printed output for the runs. A FORTRAN square-root subroutine, called from SCALE and SETPI, used an insignificant amount of time.

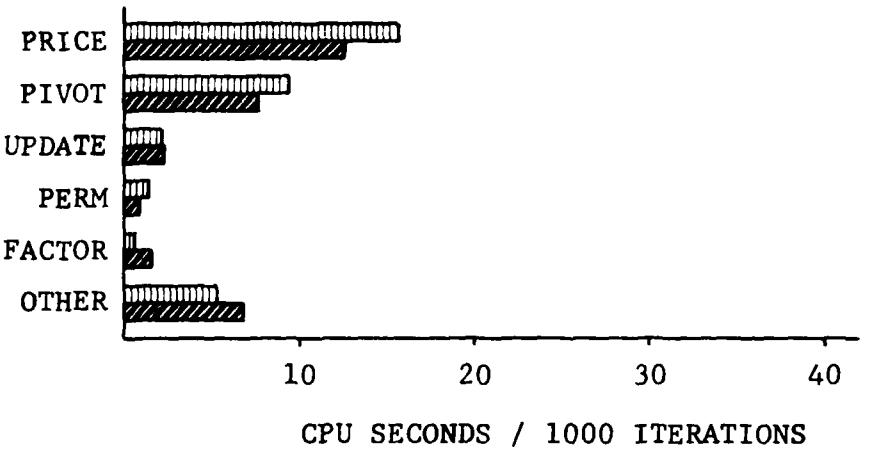
TOTAL TIME



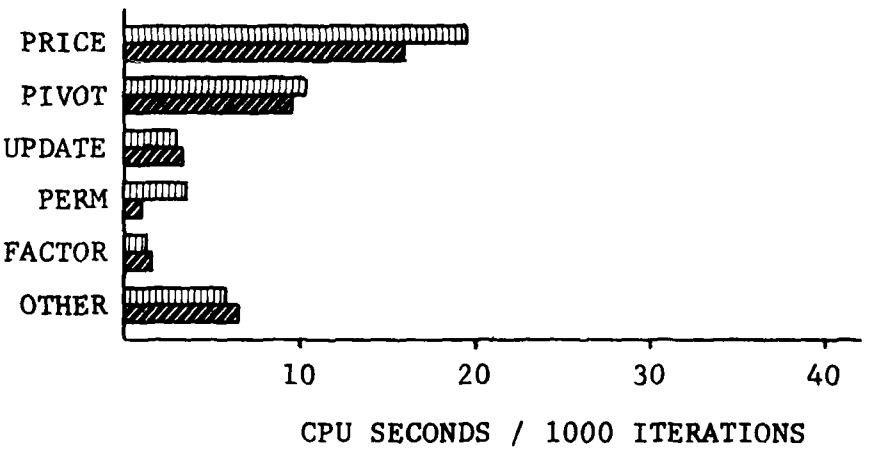
SCAGR25



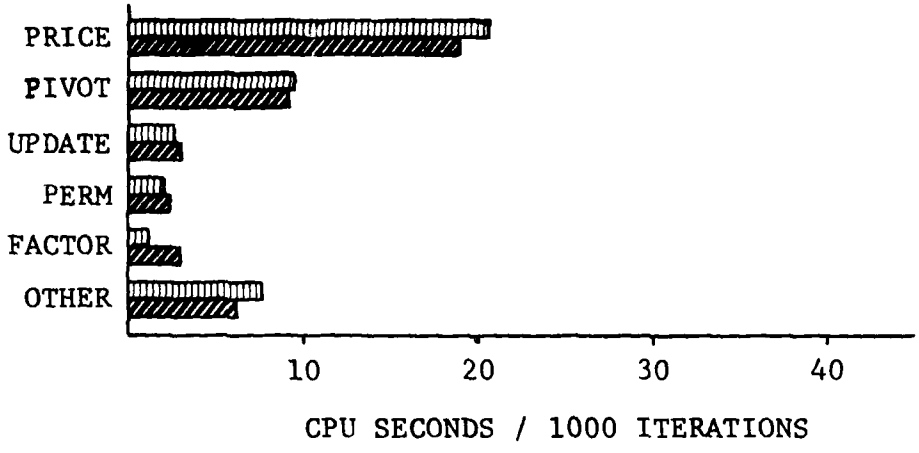
SCRS8


PRICE
PIVOT
UPDATE
PERM
FACTOR
OTHER
10
20
30
40
CPU SECONDS / 1000 ITERATIONS

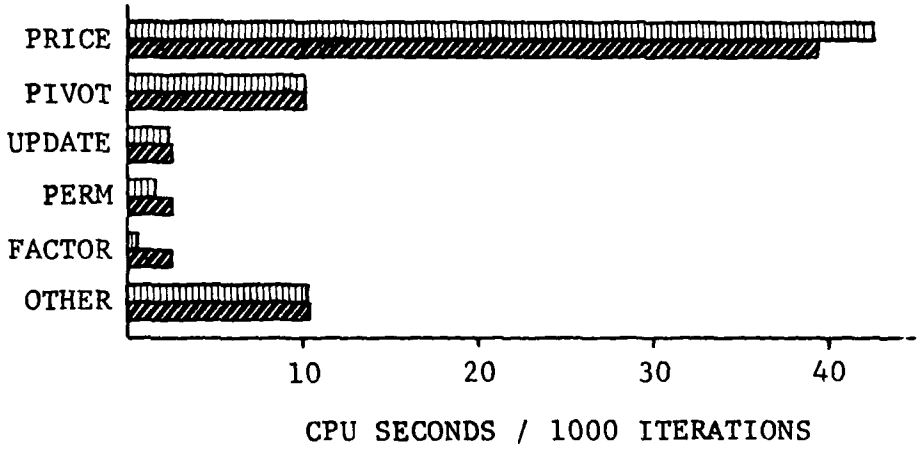
SCSD8


PRICE
PIVOT
UPDATE
PERM
FACTOR
OTHER
10
20
30
40
CPU SECONDS / 1000 ITERATIONS

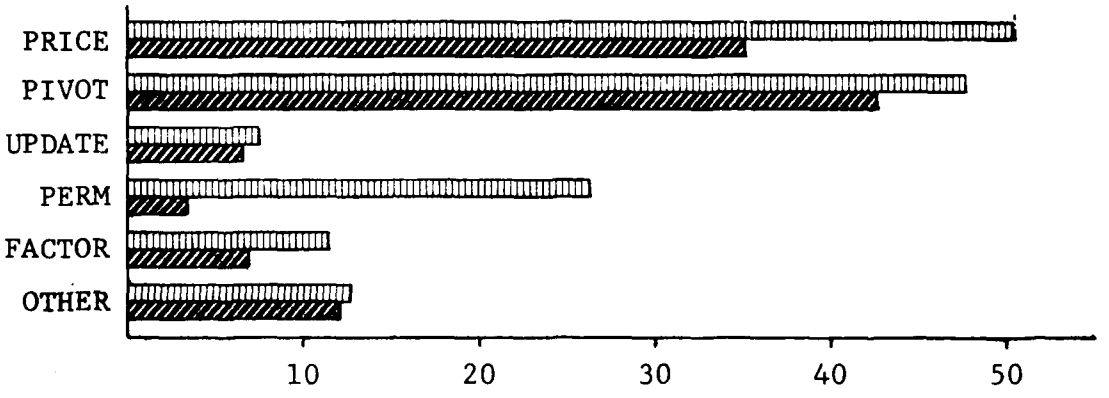
SCFXM2



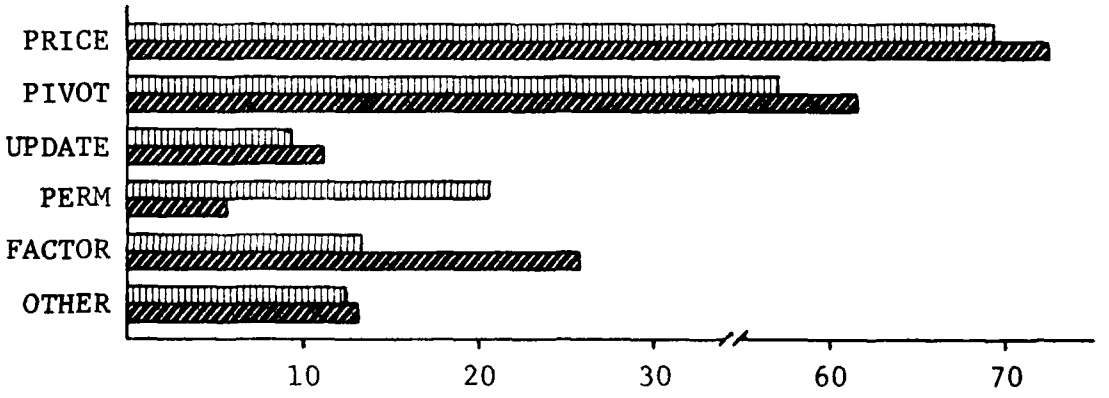
SCTAP2



PILOT


PRICE
PIVOT
UPDATE
PERM
FACTOR
OTHER
10
20
30
40
50
CPU SECONDS / 1000 ITERATIONS

BP1


PRICE
PIVOT
UPDATE
PERM
FACTOR
OTHER
10
20
30
60
70
CPU SECONDS / 1000 ITERATIONS

Iterating time

Iterating routines are those invoked at each iteration. They are classified as follows:

MAIN includes DRIVER and miscellaneous routines invoked from it: ITEROP, SETX, STATE, UNPACK and UPDBAL, and FTRANL and BTRANU when called from SETX.

PRICE refers to subroutines FORMC, PRICE and SETPI.

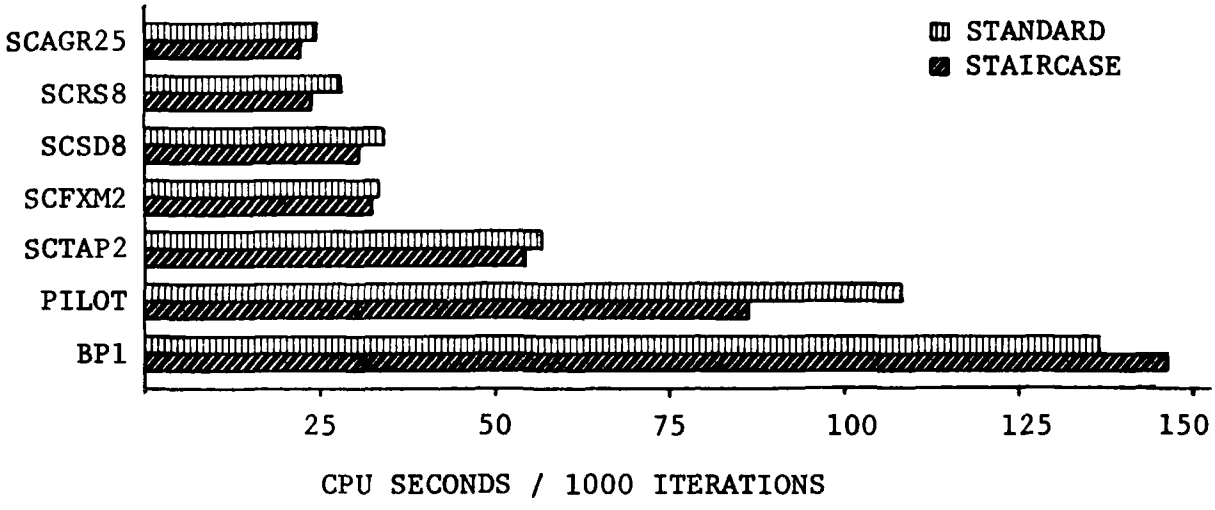
FTRANU and BTRANL refer to the like-named subroutines when called from SETPI.

PIVOT refers to subroutines LPITN and CHUZR, and UNPACK when called from LPITN.

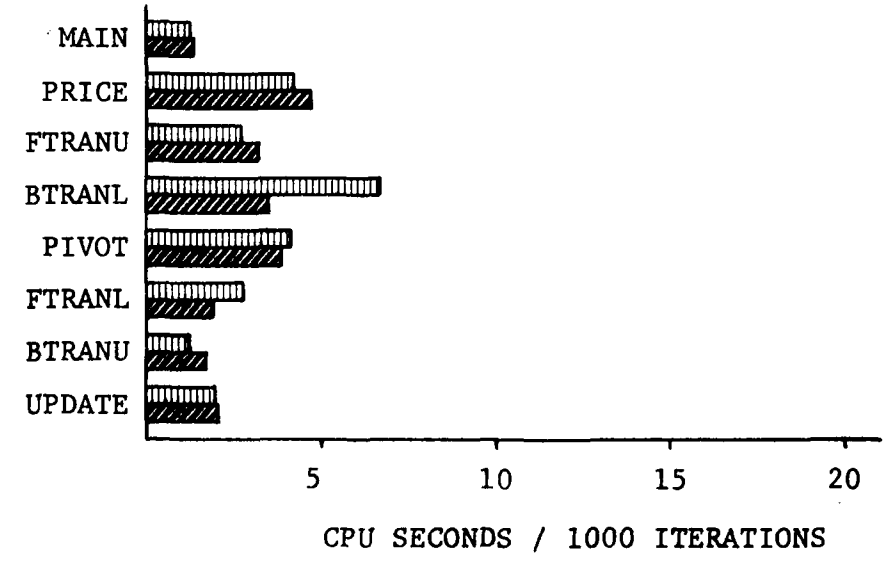
FTRANL and BTRANU refer to the like-named subroutines when called from LPITN.

UPDATE refers to subroutine MODLU.

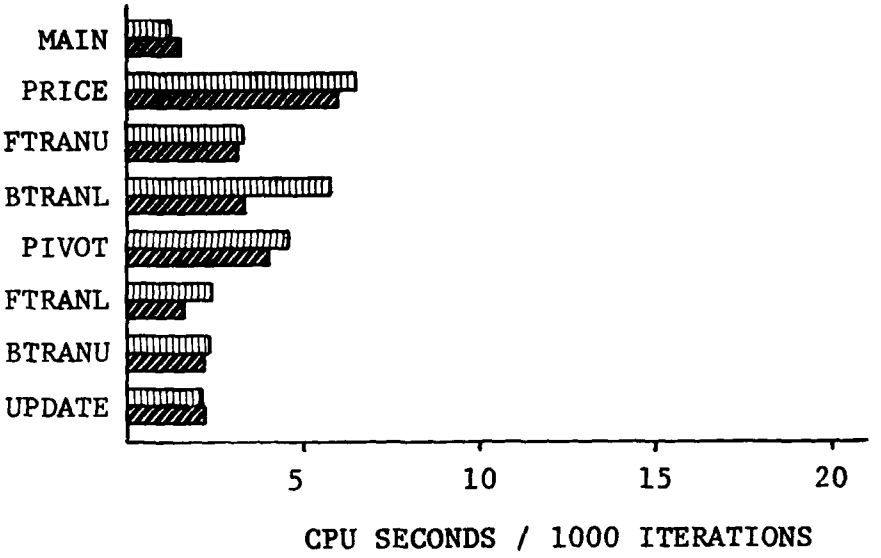
TOTAL ITERATING TIME



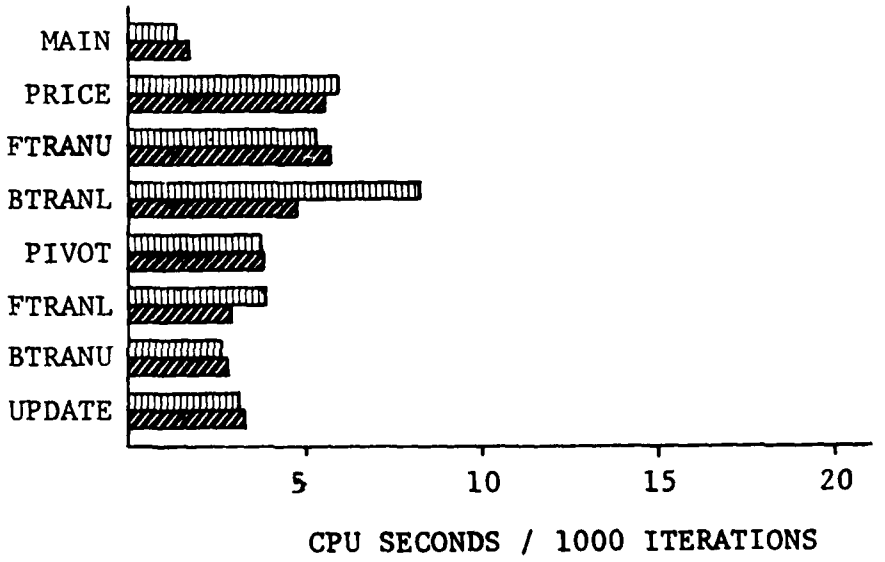
SCAGR25



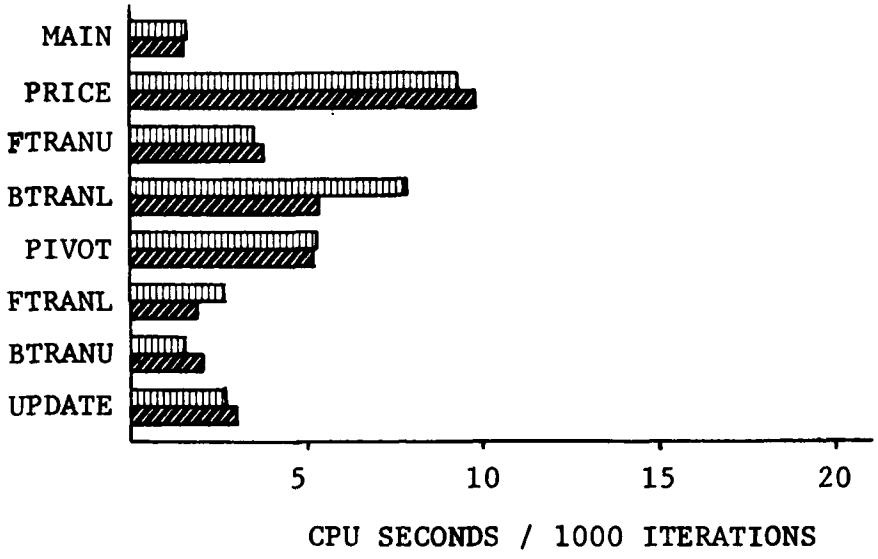
SCRS8


MAIN
PRICE
FTRANU
BTRANL
PIVOT
FTRANL
BTRANU
UPDATE
5
10
15
20
CPU SECONDS / 1000 ITERATIONS

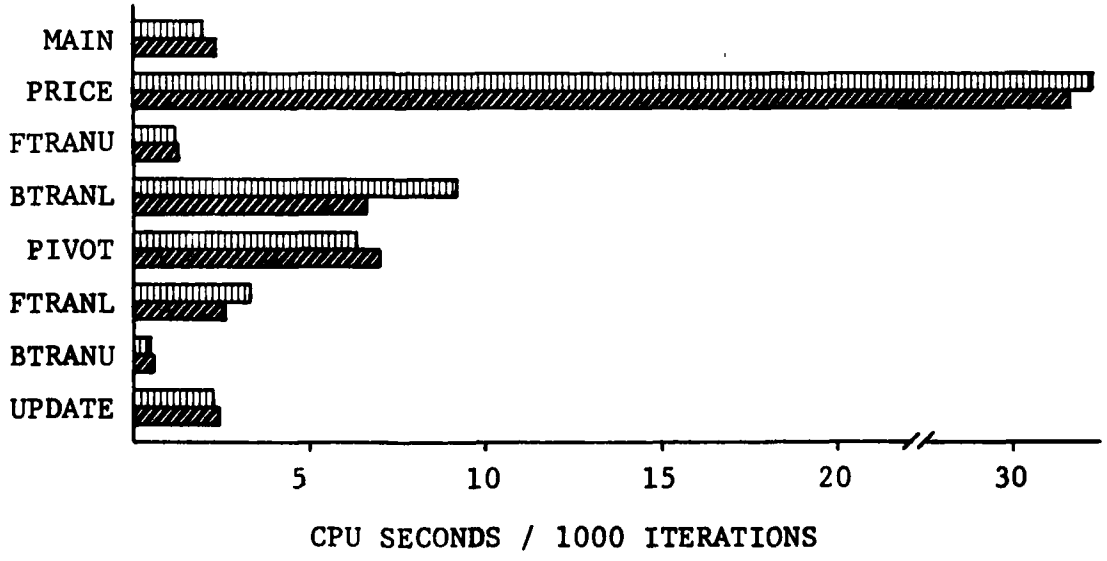
SCSD8


MAIN
PRICE
FTRANU
BTRANL
PIVOT
FTRANL
BTRANU
UPDATE
5
10
15
20
CPU SECONDS / 1000 ITERATIONS

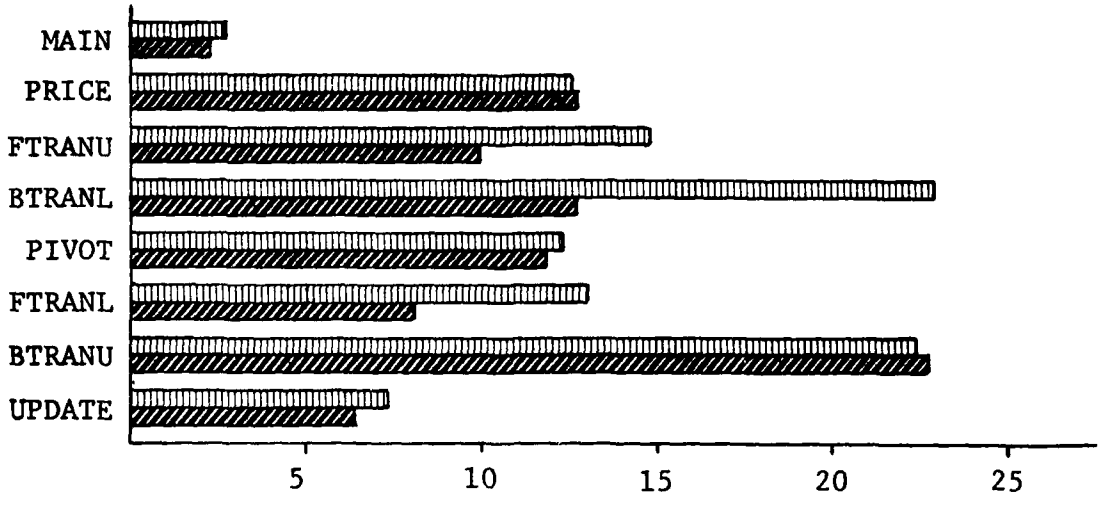
SCFXM2



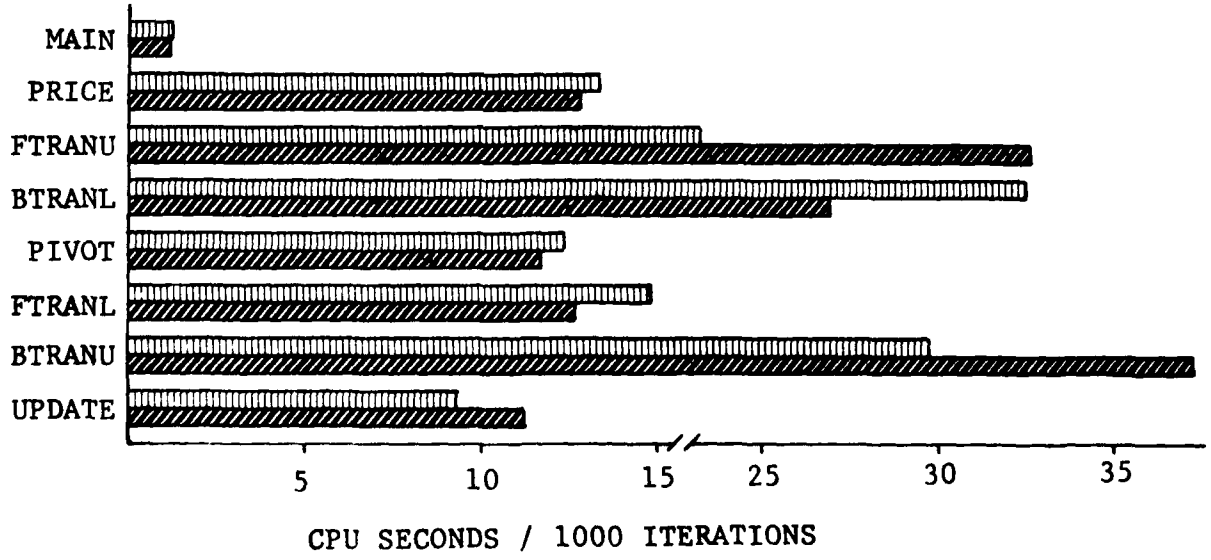
SCTAP2



PILOT


MAIN
PRICE
FTRANU
BTRANL
PIVOT
FTRANL
BTRANU
UPDATE
5
10
15
20
25
CPU SECONDS / 1000 ITERATIONS

BP1


MAIN
PRICE
FTRANU
BTRANL
PIVOT
FTRANL
BTRANU
UPDATE
5
10
15
25
30
35
CPU SECONDS / 1000 ITERATIONS

Factorizing time

Factorizing routines are those invoked at each refactorization of the basis. They are classified as follows:

MAIN includes INVERT and miscellaneous routines invoked from it: DSPSPK and SETX, and FTRANL and BTRANU when called from SETX.

PERMUTE includes the driving routine for bump-and-spike permutation--P4 with the standard method, SP4 with the staircase method--and the utility routine MKLIST.

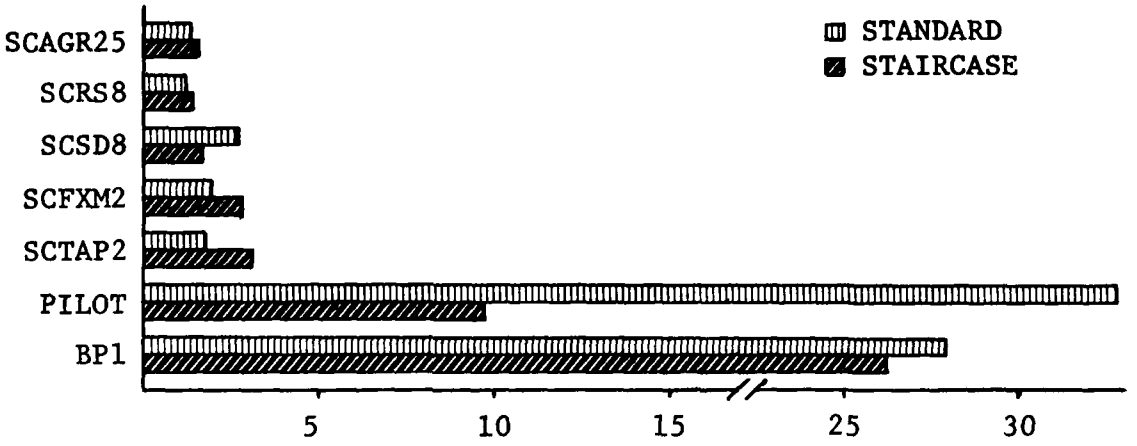
P3 refers to the subroutine that implements the spike-finding heuristic: P3 for the standard method, or SP3 for the staircase method.

BLK Δ refers to subroutines TRNSVL and BUMPS, which find a block-triangular reduction of the basis (in the standard method only).

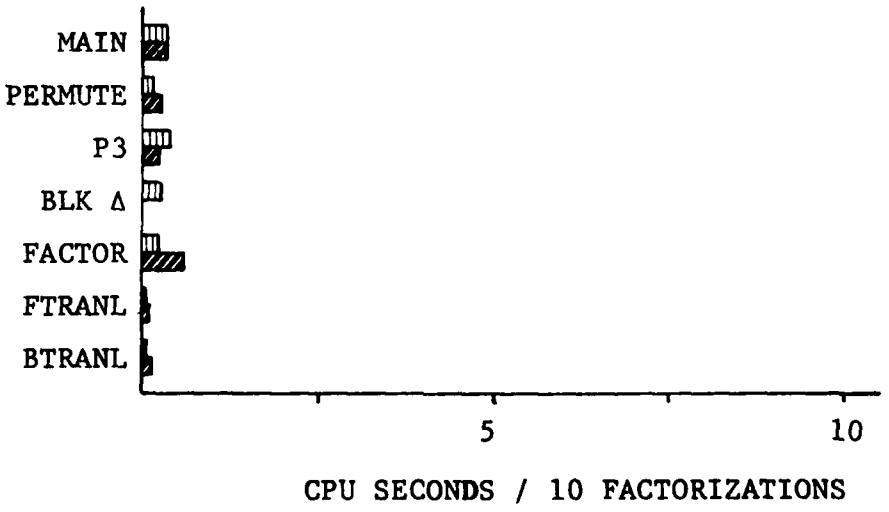
FACTOR includes subroutine FACTOR, the driving routine for LU factorization of the basis, plus routines PACKLU and UNPACK invoked from FACTOR.

FTRANL and BTRANL refer to the like-named subroutines when called from FACTOR.

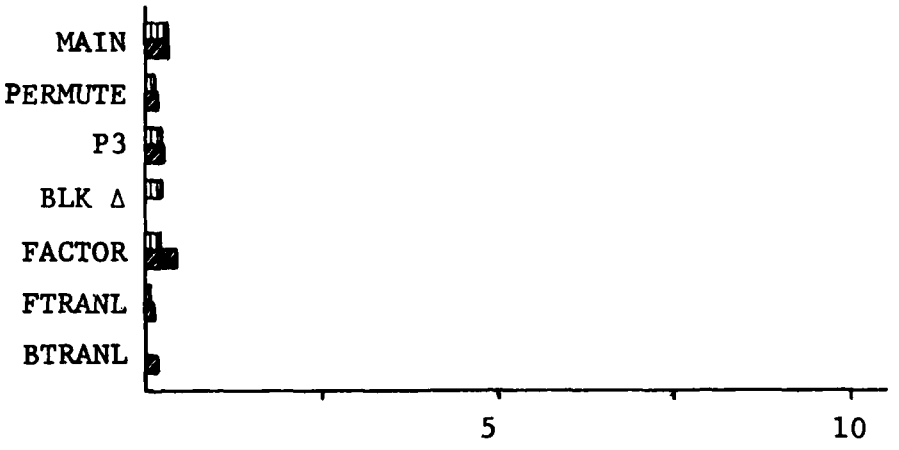
TOTAL FACTORIZING TIME



SCAGR25

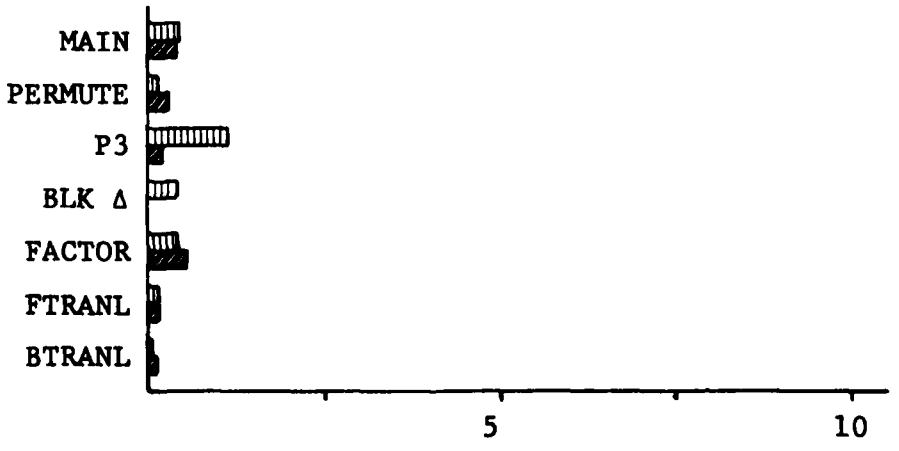


SCRS8



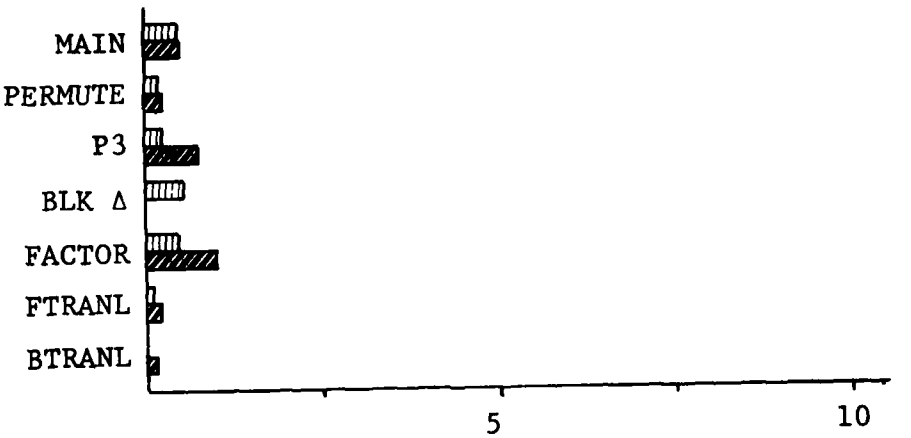
CPU SECONDS / 10 FACTORIZATIONS

SCSD8



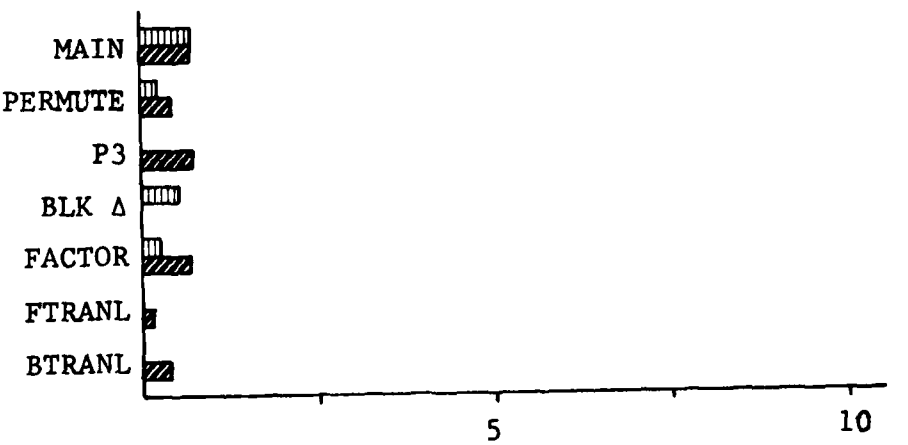
CPU SECONDS / 10 FACTORIZATIONS

SCFXM2



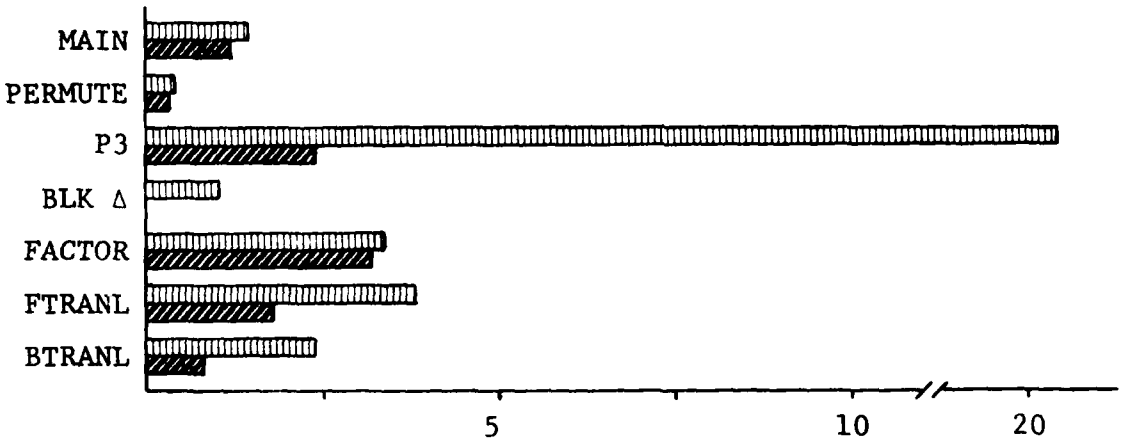
CPU SECONDS / 10 FACTORIZATIONS

SCTAP2



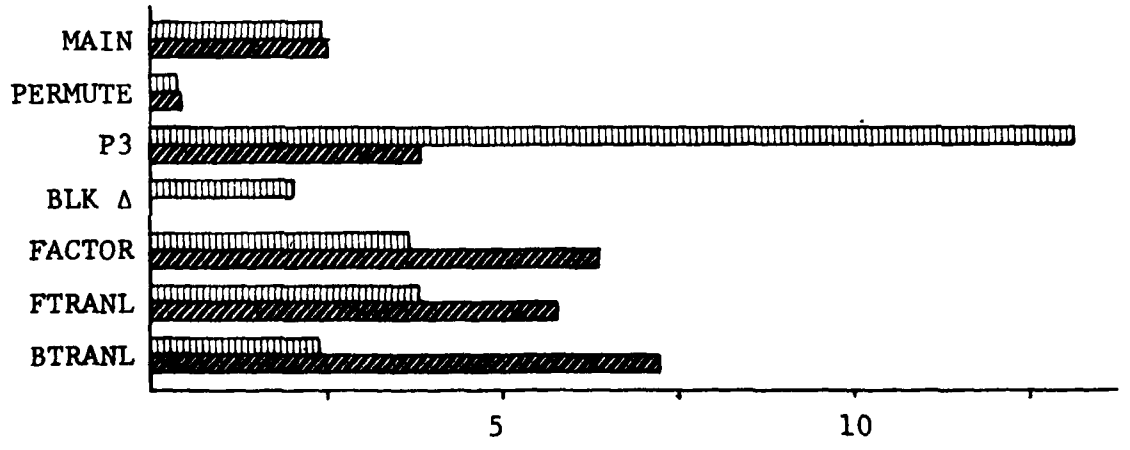
CPU SECONDS / 10 FACTORIZATIONS

PILOT



CPU SECONDS / 10 FACTORIZATIONS

BP1



CPU SECONDS / 10 FACTORIZATIONS

REFERENCES

[1] Aonuma, T., "A Two-Level Algorithm for Two-Stage Linear Programs." Journal of the Operations Research Society of Japan 21 (1978), 171-187.

[2] Bartels, Richard H., "A Stabilization of the Simplex Method." Numerische Mathematik 16 (1971), 414-434.

[3] _____ and Gene H. Golub, "The Simplex Method of Linear Programming Using LU Decomposition." Communications of the ACM 12 (1969), 266-268.

[4] Cobb, R. H. and J. Cord, "Decomposition Approaches for Solving Linked Problems." Proceedings of the Princeton Symposium on Mathematical Programming, Harold W. Kuhn, ed. (Princeton University Press, 1970).

[5] Dantzig, George B., "Programming of Interdependent Activities II: Mathematical Model." Econometrica 17 (1949), 200-211.

[6] _____, "Upper Bounds, Secondary Constraints and Block Triangularity in Linear Programming." Econometrica 23 (1955), 174-183.

[7] _____, "Optimal Solution of a Dynamic Leontief Model with Substitution." Econometrica 23 (1955), 295-302.

[8] _____, "Compact Basis Triangularization for the Simplex Method." Recent Advances in Mathematical Programming, R. L. Graves and Philip Wolfe, eds. (New York: McGraw-Hill Book Co., 1963), 125-132.

[9] _____, "Solving Staircase Linear Programs by a Nested Block-Angular Method." Technical Report 73-1, Dept. of Operations Research, Stanford University (1973).

[10] _____ and Philip Wolfe, "Decomposition Principle for Linear Programs." Operations Research 8 (1960), 101-111.

[11] Duff, Iain S., "On the Number of Nonzeroes Added when Gaussian Elimination is Performed on Sparse Random Matrices." Mathematics of Computation 28 (1974), 219-230.

[12] _____, "Practical Comparisons of Codes for the Solution of Sparse Linear Systems." Sparse Matrix Proceedings--1978, Iain S. Duff and G. W. Stewart, eds. (Society for Industrial and Applied Mathematics, 1979).

[13] _____ and J. K. Reid, "A Comparison of Sparsity Orderings for Obtaining a Pivotal Sequence in Gaussian Elimination." Journal of the Institute of Mathematics and Its Applications 14 (1974), 281-291.

[14] Forrest, J. J. H. and J. A. Tomlin, "Updated Triangular Factors of the Basis to Maintain Sparsity in the Product Form Simplex Method." Mathematical Programming 2 (1972), 263-278.

[15] Fourer, Robert, "Sparse Gaussian Elimination of Staircase Systems." Technical Report SOL 79-17, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1979).

[16] _____, "Solving Staircase Linear Programs by the Simplex Method, 2: Pricing." Technical Report SOL 79-19, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1979).

[17] Gay, David M., "On Combining the Schemes of Reid and Saunders for Sparse LP Bases." Sparse Matrix Proceedings-1978, Iain S. Duff and G. W. Stewart, eds. (Society for Industrial and Applied Mathematics, 1979).

[18] Gear, C. W. et al., "Numerical Computation: Its Nature and Research Directions." SIGNUM Newsletter, Association for Computing Machinery (1979).

[19] Glassey, C. Roger, "Dynamic Linear Programs for Production Scheduling." Operations Research 19 (1971), 45-56.

[20] _____, "Nested Decomposition and Multi-Stage Linear Programs." Management Science 20 (1973), 282-292.

[21] Goldfarb, D., "On the Bartels-Golub Decomposition for Linear Programming Bases." Mathematical Programming 13 (1977), 272-279.

[22] Grinold, Richard C., "Steepest Ascent for Large-Scale Linear Programs." SIAM Review 14 (1972), 447-464.

[23] Heesterman, A. R. G. and J. Sandee, "Special Simplex Algorithm for Linked Problems." Management Science 11 (1965), 420-428.

[24] Hellerman, Eli and Dennis Rarick, "Reinversion with the Preassigned Pivot Procedure." Mathematical Programming 1 (1971), 195-216.

[25] _____, "The Partitioned Preassigned Pivot Procedure (P^4)." Sparse Matrices and Their Applications, Donald J. Rose and Ralph A. Willoughby, eds. (New York: Plenum Press, 1972), 67-76.

[26] Ho, James K., "Optimal Design of Multi-Stage Structures: A Nested Decomposition Approach." Computers and Structures 5 (1975), 249-255.

[27] _____, "Nested Decomposition of a Dynamic Energy Model." Management Science 23 (1977), 1022-1026.

[28] _____, "A Successive Linear Optimization Approach to the Dynamic Traffic Assignment Problem." Report BNL-24713, Brookhaven National Laboratory, Upton, New York (1978).

[29] _____ and Alan S. Manne, "Nested Decomposition for Dynamic Models." Mathematical Programming 6 (1974), 121-140.

[30] IBM OS FORTRAN IV (H Extended) Compiler Programmer's Guide. No.SC28-6852, International Business Machines Corp. (1974).

[31] Johnson, R. and T. Johnston, "PROGLOOK User's Guide." User Note 33, SLAC Computing Services, Stanford Linear Accelerator Center (1976).

[32] Madsen, Oli B. G., "Solution of LP-Problems with Staircase Structure." Research Report 26, The Institute of Mathematical Statistics, Lyngby, Denmark (1977).

[33] Manne, A. S., "U.S. Options for a Transition from Oil and Gas to Synthetic Fuels." Discussion Paper 26D, Public Policy Program, Kennedy School of Government, Harvard University (1975).

[34] Markowitz, Harry M., "The Elimination Form of the Inverse and Its Application to Linear Programming." Management Science 3 (1957), 255-269.

[35] Marsten, Roy E. and Fred Shepardson, "A Double Basis Simplex Method for Linear Programs with Complicating Variables." Technical Report 531, Dept. of Management Information Systems, University of Arizona (1978).

[36] McBride, Richard D., "A Spike Collective Dynamic Factorization Algorithm for the Simplex Method." Management Science 24 (1978), 1031-1042.

[37] MPS III Mathematical Programming System: User Manual. Ketron, Inc., Arlington, VA (1975).

[38] Murtagh, Bruce A. and Michael A. Saunders, "MINOS: A Large-Scale Nonlinear Programming System (For Problems with Linear Constraints): User's Guide." Technical Report SOL 77-9, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1977).

[39] Orchard-Hays, William, Advanced Linear-Programming Computing Techniques (New York: McGraw-Hill Book Co., 1968).

[40] Parikh, S. C., "A Welfare Equilibrium Model (WEM) of Energy Supply, Energy Demand, and Economic Growth." Technical Report SOL 79-3, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1979).

[41] Perold, Andre, "Fundamentals of a Continuous Time Simplex Method." Technical Report SOL 78-26, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1978).

[42] _____ and George B. Dantzig, "A Basis Factorization Method for Block Triangular Linear Programs." Technical Report SOL 78-7, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1978).

[43] Propoi, A. and V. Krivonozhko, "The Simplex Method for Dynamic Linear Programs." Report RR-78-14, International Institute for Applied Systems Analysis, Laxenburg, Austria (1978).

[44] Reid, J. K., "A Sparsity-Exploiting Variant of the Bartels-Golub Decomposition for Linear Programming Bases." Report CSS 20, Computer Science and Systems Division, A.E.R.E. Harwell, England (1975).

[45] _____, "Fortran Subroutines for Handling Sparse Linear Programming Bases." Report AERE-R8269, Computer Science and Systems Division, A.E.R.E. Harwell, England (1976).

[46] Saigal, Romesh, "Block-Triangularization of Multi-Stage Linear Programs." Report ORC 66-9, Operations Research Center, University of California, Berkeley (1966).

[47] Saunders, Michael A., "A Fast, Stable Implementation of the Simplex Method Using Bartels-Golub Updating." Sparse Matrix Computations, James R. Bunch and Donald J. Rose, eds. (New York: Academic Press, 1976), 213-226.

[48] _____, "MINOS System Manual." Technical Report SOL 77-31, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1977).

[49] Vinson, Ilse, "Triplex User's Guide." User Note 99, SLAC Computing Services, Stanford Linear Accelerator Center (1978).

[50] Wolfe, Philip, "The Composite Simplex Algorithm." SIAM Review 7 (1965), 42-54.

[51] Wollmer, Richard D., "A Substitute Inverse for the Basis of a Staircase Structure Linear Program." Mathematics of Operations Research 2 (1977), 230-239.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SOL 79-18	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) "Solving Staircase Linear Programs By The Simplex Method, 1: Inversion"		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER SOL 79-18
7. AUTHOR(s) Robert Fourer		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0267
9. PERFORMING ORGANIZATION NAME AND ADDRESS Operations Research Department - SOL Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR-047-143
11. CONTROLLING OFFICE NAME AND ADDRESS Operations Research Program - ONR Department of the Navy 800 N. Quincy St., Arlington, VA 22217		12. REPORT DATE November 1979 13. NUMBER OF PAGES 80
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

LARGE-SCALE LINEAR PROGRAMMING
STAIRCASE LINEAR PROGRAMS
SIMPLEX METHOD

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

SEE ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SOL 79-18, Robert Fourer
"Solving Staircase Linear Programs by the Simplex Method, 1: Inversion"

Problems of economic planning, production scheduling, inventory, transportation, control and multi-stage structural design have been modeled as linear programs that have a 'staircase' structure: their activities fall into a sequence of disjoint stages or periods, while their constraints relate only successive periods. At one time it was hoped that staircase linear programs would be particularly easy to solve, owing to their special structure, but experience with the most common solution technique -- the general simplex method -- has shown otherwise. Over the years many alternatives to the simplex method have also been proposed, but as yet none of these has been proved superior in solving a wide variety of staircase problems.

This and a companion paper consider how the modern simplex method -- as implemented for large computers -- may be adapted to solve staircase linear programs more efficiently. Each paper looks at a set of algorithms within the simplex method: this one deals with 'inversion' of the basis -- more accurately, solution of linear systems by Gaussian elimination -- and its successor considers the task of 'pricing'.

Both papers describe extensive (though preliminary) computational experience, and can point some quite promising results.